

Module - I : Core Java PROGRAMMING

Section 1: Basics of Programming.

- * Software Application
- * JavaC, JVM, JRE, JDK
- * Compilations & Execution
- * Variables & operators
- * Control statements
- * Functions
- * Arrays & String

Section 2: OOPS

* Class & Objects

* Blocks

* Constructors

* Encapsulation

* Has A relationship

* Is A relationship (Inheritance)

* Method overloading & overriding

* Type Casting

* Polymorphism

* Abstract class

* Interfaces

* Abstraction

* Java Bean class

* Singleton class

Section 3: JAVA LIBRARIES.

* Object class functions

* String functions

* Exception Handling

* Collection Framework Library

* Threads in JAVA

* File Programming.

MODULE 2: J2EE

* Java Programming

* SQL

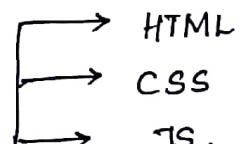
Section 1: JDBC Programming

* insert record into DB

* retrieve record from DB

* update record in DB

* delete record from DB.



↳ Web Technologies

Section 2: Servlets Programming

↳ JDBC programming

* Java Programs to run on Server.

→ JSP Programming

↳ MVC Architecture

Section 3:

FRAMEWORKS: Module 3 ↳ Spring.

↳ Hibernate

Module 4:

Android

Development

** SOFTWARE APPLICATION:

* Develop for business needs or business requirements.

Ex: Banking application.

* Programs are written using a Software language.

* Softwares are used to eliminate manual works.

** TYPES OF SOFTWARE APPLICATION:

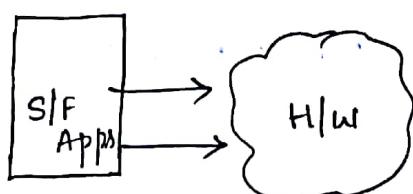
* System Software.

* Application Software.

** SYSTEM SOFTWARE:

→ Software application build to interact with Hardware.

→ Programs run with the help of Hardware functionalities.



Languages: C, C++.

CPU / Memory / Peripherals.

Examples: OS, compiler, Assemblies, Device drivers, chip drivers

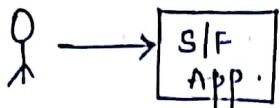
** APPLICATION SOFTWARE:

Software

→ Application built to reduce human intervention or to

fulfil end user needs.

Examples: Gmail



Facebook

Book My Show

Flipkart

→ These are developed using Java, .Net, PHP, SAP.

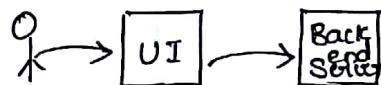
** 3 COMPONENTS OF APP/SF.

* FRONT END OR USER INTERFACE:

→ User interacts with software applications with UI.

→ Software languages are used to build UI.

→ UI interacts with Backend.



* BACKEND OR MIDDLE LAYER OR BUSINESS LOGIC LAYER.

→ write programs to do all business related operations.

→ Business operation depends on project.

DATA BASE:

→ Used to store information / data.

→ Data.

↳ → User Provided data. → Phone No, Email Id

↳ → Appn generated data. → OTP, Acknowledgement.

** S/F
 -- Application Architecture:

1) Stand Alone application.

2) Client-Server application.

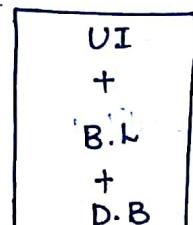
3) Web-based application.

4) Mobile based application.

* Stand alone

Application: Single User Application.

Ex: Antivirus & OS.



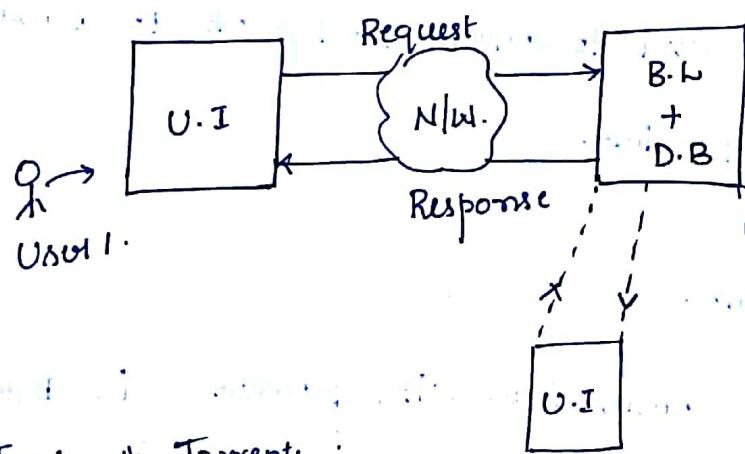
* Advantages:

- Security
- Performance

* Disadvantages:

- Multiple user cannot access d.
- NO sharing of data
- No remote access.

* Client - Server Application:



→ Two-tier architecture

classification of clients

* Thin & fat clients.
 * Thick clients.

Ex: 1) Torrents

2) Skype

3) ATM

* Advantages:

- - - - -

* Multi User

- - - - -

* Sharing Information

- - - - -

* Remote Access

- - - - -

* Disadvantages

- - - - -

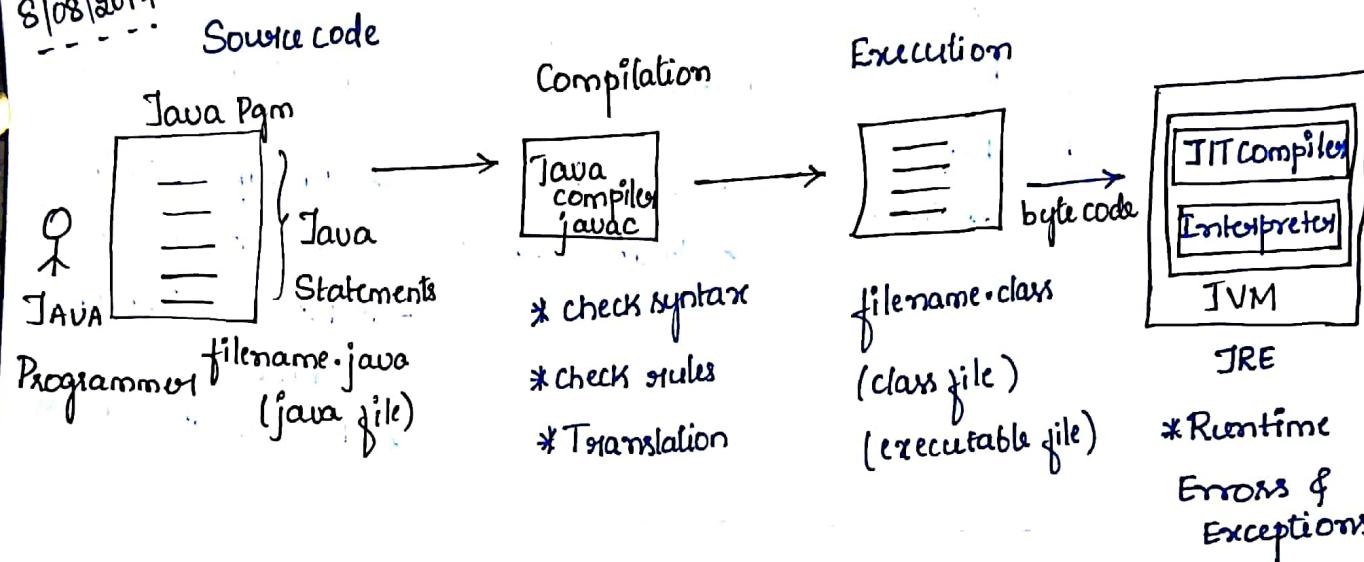
Security

Performance

Networks

Availability of Server.

8/08/2017



JIT : Just-In Time Compiler

This compiler is used to convert the bytecode to a machine level language at the runtime.

JVM : JAVA Virtual Machine.

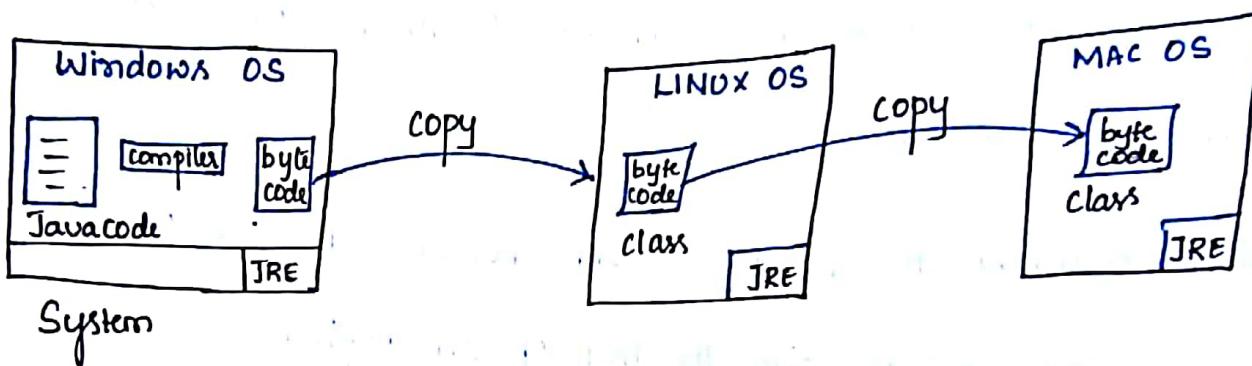
This is used to communicate with processor. ie Bytecode generated after compilation is not in the MLL. The processor can't understand, Bytecode is called a intermediate code.

JRE : JAVA Runtime Environment (Provides an environment for execution).

Runtime execution takes more time compared to C for execution.

Because, at the execution time Java compiler (JIT compiler) then interprets. This is done to achieve Platform Independence.

** PLATFORM INDEPENDENT



JAVA is platform independent, as it runs on any operating system with JRE.

Developing JAVA programs involves 3 steps :

- * Source code creation.
- * compilation.
- * Execution.

* Source Code creation:

The java program is created and saved in a file with the extension .java. This file contains only Java Statements.

* Compilation Stage:

The source code is converted into byte code by JAVA compiler

- The JAVA compiler translates the JAVA statements into byte code statement and save it in a file with extension .class.
- The .class file is generated with JAVA compiler.
- The bytecode statement is not in a machine executable format.

* Execution Stage:

- In execution stage, the JVM executes the bytecode and gives the result of the bytecode.
- While executing the bytecode, JVM translates the bytecode into Machine executable format with the help of JIT compiler.
- The O/P of the JIT compiler is interpreted by the JVM to give the result of program.

* The .class file can be executed on any OS provided JRE's installed on the system. This concept is known as Platform Independence.

→ We cannot run the .class file without JRE. Hence, it is dependent on JRE.

* When compiling if we get any error, it is known as Compile Time error. Until we fix compile time error, compiler will not generate .class file.

→ Any error occurring at execution time is known as Runtime error.
To fix the runtime error, we have to goto source code do the required changes, recompile it and then execute it.

** JDK → JAVA Development Kit.

JDK contains Java compiler & JRE.

& also few development tools needed for developing JAVA applications.

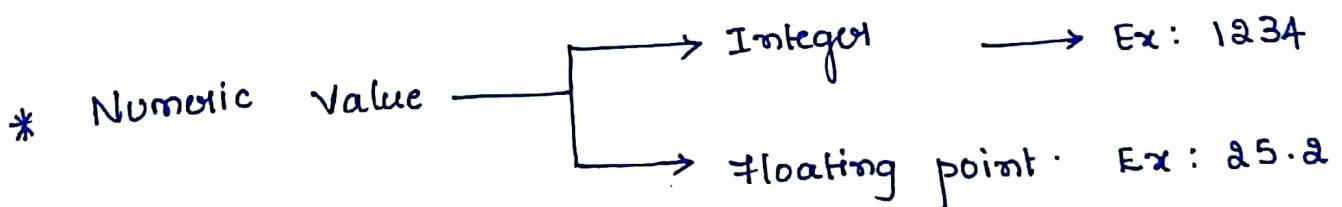
* IDE → Integrated Development Environment or Text Editor is used to generate Source code.

Ex: Notepad, Notepad++, Editplus, Eclipse, Netbeans.

glossary:

** JAVA BASIC PROGRAMMING

* TYPE OF VALUES



* Character Value

It is enclosed in ''.

Eg: 'q', 'j' 'i', '@'.

* Boolean Value

Represented with keywords true or false.

* String Value

Enclosed with " "

Eg: "jspiders", "1234", "ab1a3".

→ In java there is no need for type specifiers.

Program 1:

```
>     class Program1  
>     {  
>         public static void main (String [] args)  
>         {  
>             System.out.println ("Hi, welcome to Jspiders");  
>         }  
>     }  
> - - - - - - - - - - - - -
```

O/P: Hi, welcome to Jspiders.

Program 2:

```
>     class Program2  
>     {  
>         public static void main (String [] args)  
>         {  
>             System.out.println ('12345');  
>             System.out.println (123.45);  
>             System.out.println ("Hi");  
>             System.out.println ('j');  
>             System.out.println (true);  
>         }  
>     }
```

O/P: 12345
123.45
Hi
j
true

`System.out.println (value)`

Prints value, "move cursor to nextline".

Ex: `System.out.println ('h');`

`System.out.print ('h');`

`System.out.println ('i');`

`System.out.print ('i');`

O/P: h

i
+ → Cursor

O/P: h i

↓
Cursor.

10/08/2017

* OPERATORS

+ Operator: In java + Symbol is used in two ways.

* Addition.

$$20 + 20 = 40$$

* Concatenation

"java" + "developer" = javadeveloper.

JAVA does not support operator overloading except '+' operator. (operator performing more than one function is called operator overloading).

Program:

Class Program

{

`PSVM (String [] args)`

```
SOP (20+20);  
SOP ("JAVA" + "developer");
```

O/P: 40
JAVAdeveloper.

Program 6:

```
class Program6  
{  
    PSVM (String [] args)  
{  
        SOP ("Number is" + 20);  
        SOP ("Number is " + 20 + 20);  
        SOP ("Number is " + (20+20));  
        SOP (20+ " is the number");  
        SOP (20+20 + " is the number");  
    }  
}
```

O/P:
Number is 20
Number is 2020
Number is 40
20 is the number
40 is the number

Program 7: Addition of ASCII Values

```
class Program7  
{  
    PSVM (String [] args)  
{
```

ASCII values
a = 97
b = 98
A = 65
B = 66

O/P: -----

```
    SOP ("jspider" + 's');  
    SOP ('a' + 'b');
```

jspiders

195

// Addition of ASCII values.

```
}
```

'\n' → new line .

'\t' → tab space .

Program 8 :

```
class Program 8
{
    PSVM (String [] args)
    {
        SOP (" Ramesh \t Kumar \t Sharma ");
        SOP (" Ramesh \n Kumar \n Sharma ");
    }
}
```

O/P:

Ramesh Kumar Sharma

Ramesh

Kumar

Sharma

** String Value :

----- → Anything written inside double quotes
" " if taken as string.

↑ ↑
Open of String closing of String.

back slash → Back slash \ is used to escape the meaning
of next character .

Example : Program 9

```
class Program9
{
    PSVM (String [] args)
    {
        SOP ( " I am a \" Software \" Engineer ");
    }
}
```

O/P:

I am a " Software " Engineer.

** KEYWORDS & IDENTIFIERS

Keywords are predefined words provided by the language itself.

→ They are the reserved words used for particular purpose.

→ In JAVA language all the keywords are mentioned in lower case.

Ex: public, static, final, abstract etc

Identifiers are the names provided by the programmers to identify the elements of programs.

We use Identifiers to identify

- * Variable
- * Function
- * class name
- * package.

→ When programmers provide identifiers should obey the rules of identifiers.

1) The Identifier must begin with alpha character.

2) The Identifier can be a combination of alpha-numeric values but the first letter should be alpha character.

3) We can use underscore ⁽⁻⁾ or [special character] \$ (dollar)

4) Keywords are not allowed as identifiers.

5) No spaces [between]/allowed in the identifiers

Valid Identifiers

empid
empid123
emplazid
emp-id

Invalid Identifiers

lempid
emp id
emp@id
abstract.

** VARIABLES :

→ Used to store values.

Variable declaration : datatype . variablename ;

Variable Initialization : variablename = value ;

datatype variablename = value ;
 ↓ ↓
 Type of values Identifier .

Type of values

	Data Type	Default value	Default Size
Integer value	Byte	0	1
	Short	0	2
	int	0	4
	long	0L	8
floating point value	float	0.0f	4
	double	0.0d	8
Character value	char	'\u0000'	2
Boolean value	boolean	true false	1bit

"String" is not a Datatype.

Example for Declaration of Variables

```
int empid;
double empsalary;
char empgrade;
```

Example for Variable Initialization

```
empid = 4189;
empsalary = 25000.00;
empgrade = 'A';
```

Or

```
int empid = 4189;
double empsalary = 25000.00;
char upgrade = 'A';
```

Programs on Variables:

```
class Program1
{
    public static void main (string [] args)
    {
        SOP ("Program Started");
        // variable declaration
        int empid;
        double empsalary;
        char empgrade;
    }
}
```

//variable initialization.

```
empid = 2451 ;  
empsalary = 2500.00 ;  
empgrade = 'A' ;
```

```
SOP ("Employee ID is " + empid);  
SOP ("Employee Salary is " + empsalary);  
SOP ("Employee Grade is " + empgrade);  
SOP ("program ended");  
}
```

O/p:

Employee ID is 2451
Employee Salary is 2500.00
Employee Grade is A.

NOTE:-

All local variables must be initialized before using it in any operations otherwise the compiler throws an error.

class Programs

```
{  
    public static void main (String [] args)  
    {  
        int p=12; double q=1.1; int r=3;  
        int x,y,z;  
        int f=12, j=34, k=45;  
    }  
}
```

**

```
int p = 12, double q = 10.0, int r = 5;
```

**

This is not possible in JAVA.

14/08/2017

```
class Program3
{
    public static void main (String [] args)
    {
        System.out.println ("Program Started");
        int x = 12; // declaration & initialization
        SOP (x);
        x = 34; // re-initialization
        SOP (x);
        x = 45; // re-initialization
        SOP (x);
        System.out.println ("Program ended");
    }
}
```

O/P:
12
34
45

class Program4

```
{
    public (String [] args)
    {
        int x = 12;
        int y = 23;
        int z = 45;
        SOP ("x value is " + x);
        SOP ("y value is " + y);
        SOP ("z value is " + z);
    }
}
```

```
y = x; // copy value of x to y.
```

```
x = y; // copy value of y to x.
```

```
SOP ("The value of x is " + x);
```

```
SOP ("The value of y is " + y);
```

```
SOP ("The value of z is " + z);
```

O/P:-

x value is 12

y value is 23

z value is 45

The value of x is 12

The value of y is 12

The value of z is 12.

** When specifying or initializing long and float numbers directly in the program, specify or add suffix to the declaring number

class Programs

```
{
```

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

```
    SOP ("Pgm started");
```

```
    long phnum = 7892227315L;
```

```
    float marks = 80.01f;
```

```
    SOP (phnum);
```

```
    SOP (marks);
```

O/P:-

7892227315

80.01

** Any variable declared as final, we should not re-initialize.

If we re-initialize, compilation error occurs **

Example:

```
class Program6
{
    public static void main (String [] args)
    {
        int final _stid = 15;
        System.out.println (_stid);
        _stid = 16; // error, cannot reinitialize final variable.
    }
}
```

→ If any variable declared with final keyword we call that variable as final variable in other words constant.

→ Final variables must be initialized only 1 time.

→ If we try to initialize, compiler throws an error.

** OPERATORS:

Simple answer can be given using expression $4+5$ is equal

to 9. Here 4 and 5 are called operands.

& + is called operator.

Class Program

```
{  
    public static void main (String [] args)  
    {  
        int x = 10; int y = 4; int z = 2;  
        int r1, r2, r3, r4, r5;  
  
        r1 = x + y ;  
        r2 = x - y ;  
        r3 = x * z ;  
        r4 = x / z ; // Quotient value.  
        r5 = x % z ; // Remainder value .  
  
        System.out.println (r1); System.out.println (r2);  
        System.out.println (r3); System.out.println (r4);  
        System.out.println (r5);  
    }  
}
```

O/P:

14

10

20

5

0

**

Operator operations always takes place from left to Right.

$$\text{Ex: } 2 * 3 / 2 \Rightarrow 6 / 2 \Rightarrow 3.$$

**

JAVA supports following type of operators.

1) Arithmetic operators.

2) Comparison operators.

3) Logical or Relational operators.

4) Assignment Operators.

5) Conditional or Ternary operators.

** ARITHMETIC OPERATORS:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.

Assume integer variable, A holds 10 & variable B holds 20.

Operator	Description
+	Addition → Adds values on either side of the operator $A+B$ gives 30.
-	Subtraction → Subtracts right hand operand from left hand operand. $A-B$ gives -10.
*	Multiplication → Multiplies values on either side of the operator. $A*B$ gives 200.
/	Division → Divides left hand operand by right hand operand. B/A gives 2.
%	Modulus → Divides left hand operand by right hand operand and returns remainder. $B \% A$ gives 0.
++	Increment → Increases the value of the operand by 1. 1. $B++$ gives 21.
--	Decrement → Decreases the value of operand by 1. $B--$ gives 19.

** RELATIONAL OPERATORS :

The relational operations determine the relationship that one operand has to the other.

Assume variable A holds 10 & variable B holds 20, then.

Operator	Description.
$= =$	→ Checks if the values of two operands are equal or not, if yes, then the condition becomes true. $(A == B)$ is not true.
$!=$	→ Checks if the values of two operands are equal or not, if not equal then condition becomes true. $(A != B)$ is true.
$>$	→ Checks if the value of left hand operand is greater than the value of right operand, if yes the condition becomes true. $(A > B)$ is not true.
$<$	→ Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. $(A < B)$ is true.
\geq	→ Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. $(A \geq B)$ is not true.

\leq → checks if the value of left hand operand is less than or equal to the value of right operand, if yes then condition becomes true.

$(A \leq B)$ is true.

* * BITWISE OPERATORS:

JAVA defines several bitwise operators, which can be applied to the integer types, long, int, short, byte and char.

→ Bitwise operators works on bits and performs bit-by-bit operation.

Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands.
	Binary OR Operator copies a bit if it exists in either operand.
\wedge	Binary XOR operator copies the bit if it is set in one operand but not both.
\sim	Binary Ones Complement Operator & has the effect of 'flipping' bits.
\ll	Binary Left Shift Operator. The left operands value moved left by the number of bits specified by the right operand.

\gg → Binary Right Shift Operator
 $(>>)$

The left operand's value is moved right by the number of bits specified by the right operand.

$>>>$ → Shift Right Zero Fill Operator.

The left operand's value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.

Assume if $a = 60$ & $b = 13$; now in binary format they will be as follows :

$$a = 0011\ 1100$$

$$b = 0000\ 1101$$

$$(a \& b) = 0000\ 1100$$

$$(a | b) = 0011\ 1101$$

$$(a ^ b) = 0011\ 0001$$

$$(\sim a) = 1100\ 0011$$

$a \ll 2$ will give 240 which is 1111 0000

$a \gg 2$ will give 15 which is 1111

$a \gg> 2$ will give 15 which is 0000 1111

** LOGICAL OPERATORS

A logical operator in JAVA programming is an operator that returns a Boolean result that is based on the Boolean result of one or two other expressions.

Operator Description.

`&&` → called LOGICAL AND operator. If both the operands are non-zero, then the condition becomes true.
 $(A \&\& B)$ is false.

`||` → called LOGICAL OR Operator. If any of the two operands are non-zero, then the condition becomes true.
 $(A || B)$ is true.

`!` → called LOGICAL NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.
 $!(A \&\& B)$ is true.

** ASSIGNMENT OPERATORS:

An assignment operator is the operator used to assign a new value to a variable, property, event etc in JAVA.

Operator	Description.
$=$	→ Simple assignment operator, Assigns values from right side operands to left side operand. $C = A+B$ will assign value of $A+B$ into C .
$+ =$	→ Add AND assignment operator, It adds right operand to left operand and assign the result to left operand. $C += A$ is equivalent to $C = C + A$.
$- =$	→ Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand. $C -= A$ is equivalent to $C = C - A$.
$* =$	→ Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand. $C *= A$ is equivalent to $C = C * A$.
$/ =$	→ Divide AND assignment operator, It divides left operand with the right operand & assign the result to left operand. $C /= A$ is equivalent to $C = C / A$.
$\% =$	→ Modulus AND assignment operator, it takes modulus using two operands and assign the result to left operand. $C \% = A$ is equivalent to $C = C \% A$.
$<< =$	→ Left shift AND assignment operator $C \ll= 2$ is same as $C = C \ll 2$.

$>>=$ → Right Shift AND assignment operator.
 $c >>= 2$ is same as $c = c >> 2$.

$\&=$ → Bitwise AND assignment operator.
 $c \&= 2$ is same as $c = c \& 2$.

$!=$ → Bitwise inclusive OR and assignment operator.
 $c != 2$ is same as $c = c != 2$.

$^=$ → Bitwise Exclusive OR and assignment operator.
 $c ^= 2$ is same as $c = c ^ 2$.

* CONDITIONAL OPERATOR:

Conditional operator is also known as Ternary Operator.
This operator consists of three operands and is used to evaluate Boolean expressions.

The goal of the operator is to decide which value should be assigned to the variable. The operator is written as

variable $x = (\text{expression}) ? \text{value if true} : \text{value if false}$

→ Following is the example.

```
public class Test  
{
```

```

public static void main (String [] args)
{
    int a, b;
    a = 10;
    b = (a == 1) ? 20 : 30;           O/P:
    SOP ("value of b is : " + b);     value of b is : 30
    b = (a == 10) ? 20 : 30;          value of b is : 20
    SOP (" value of b is : " + b);
}

```

** PRECEDENCE OF JAVA OPERATORS:

Category	Operator	Associativity.
Postfix	{ } [] .	left to Right
Unary	++ -- ! ~	Right to Left
Multiplicative	* / %	left to Right
Additive	+ -	left to Right
Shift	>>> <<	left to Right
Relational	> >= < <=	left to Right
Equality	= !=	left to Right

Bitwise AND	&	Left to Right
Bitwise XOR	^	Right to Left
Bitwise OR		Right to Left
Logical AND	&&	Right to Left
Logical OR		Right to Left
Conditional	? :	Right to Left

Assignment
 $=, +=, -=, *=,$
 $/=, \% =, >> =, <= =$ Right to Left
 $\&=, ^=, |=$

Comma , Left to Right

* INSTANCE OF OPERATOR

This operator is used only for reference variables. It

operator checks whether the object is of a particular type (class type or interface type).

Instance of operator is written as

(object reference variable) instance (class / interface type)

** UNARY OPERATORS

- Works on only one data.
- $++$ → increment operator always increments value by 1.
- $--$ → decrement operator always decrements value by 1.

Example: int $x = 1;$

$$x++ \Rightarrow x = x + 1.$$

Or

int $x = 1$

$$x-- \Rightarrow x = x - 1.$$

Post

Pre.

$x++$

$++x \leftarrow$ Pre-increment.

$x--$

$--x \leftarrow$ Pre-decrement.

Example:

int $x = 0$

$x++$

SOP(x): $x = \underline{1}$

$++x$

SOP(x):

$x = \underline{2}$

$x--$

SOP(x):

$x = \underline{1}$

$--x$

SOP(x):

$x = \underline{0}$

15 Aug 2017

Example:

```
int x = 0;  
int y = 0;  
y = x++;  
SOP(x);  
SOP(y);
```

Ans:

0	1	1	0
1	0	1	0

y = x++; → Use current value in operation, later increment the variable.

Example 2:

```
int x = 0;  
int y = 0;  
y = ++x  
SOP(x);  
SOP(y);
```

Preincrement, increment the value first, use the value in operation.

O/P:

1

1

Example 3:

```
int x = 0;  
int y = 0;  
y = x++ + x;  
SOP(x);  
SOP(y);
```

y = 1 +

$$\begin{aligned}y &= x + x \\y &= 0 + 1 \\&\quad x = x + 1 \\y &= 1 \quad x = 1\end{aligned}$$

O/P:

1

y = x++

y = 0

x++ = x + 1
x = 1

Ex 4: int $x = 0$

int $y = 0$

$y = x++ + x++ + x;$

SOP(x); // 2

SOP(y); // 3.

O/P:

$y = 0 + 1 + 2$

$y = 3$

$x = 2$

Ex 5:

int $x = 0$

int $y = 0$

$y = x + x + x++;$

SOP(x); // 1

SOP(y); // 0

O/P:

$y = 0 + 0 + 0$

$y = 0$

$x = 1$

$x = x + 1$

$x = 0 + 1$

Ex 6:

int $x = 0$

int $y = 0$

$y = x^{^0} + x^{^1} + x^{^2} + x^{^3} + x^{^4} + x^{^5} + x^{^6};$

SOP(x);

$x = x + 1$
 $x = 1$

SOP(y);

$x = 2$

$y = 0 + 1 + 2 + 3 + 4 + 5 + 6$

$x = 3$

$y = 21$

Ex 7:

int $x = 0$

int $y = 0;$

$y = ++x + x + ++x;$

SOP(x); // 2

SOP(y); // 4

$\longrightarrow 1 + 1 + 2$

O/P

2

4.

Ex 8: int $x = 0$; int $y = 0$;

$y = ++x \oplus x++ \oplus ++x \oplus x \oplus x++ \oplus ++x \oplus ++x;$

SOP(x); // 6

O/P: 6

SOP(y); // 22

22

Ex 9: int $x = 0$; ***

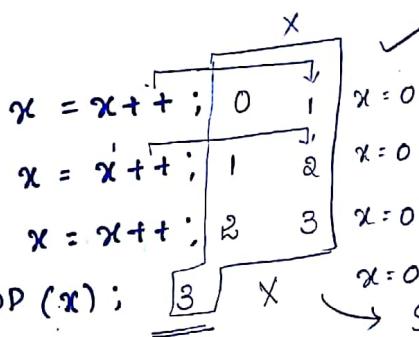
$x = ++x$; 1

$x = ++x$; 2

$x = ++x$; 3

SOP(x); // 3

Ex 10: int $x = 0$;



Assignment operator
is given the higher
priority.

Ex 11: int $x = 1$; int $y = 0$;

$y = x - 1$;

SOP(x); SOP(y);

// 0

// +1

Ex 12

$y = -x$

SOP(x); SOP(y);

// 0; // 0;

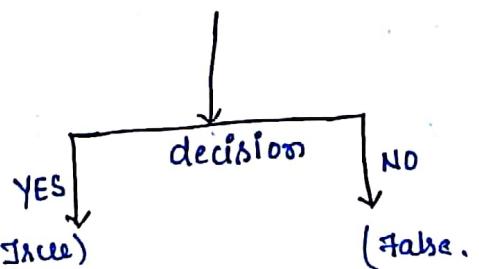
** CONTROL STATEMENTS:

→ are used to control flow of execution.

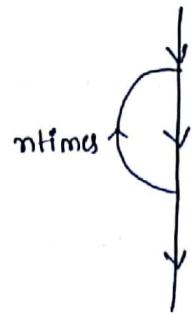
a) if statement

b) Switch Statement (true)

if Branching.



2) Looping.



or for statement

or while statement.

↳ do-while statement.

** if Statement:

Syntax:

Result in
boolean
value

```
if (condition)
  Stmt 1 ;
  Stmt 2 ;
  Stmt 3 ;
}
```

Relational
or
Logical
operator.

→ The statements written in the body of if are executed only when the condition is resulting in true otherwise the statements of if body will not be executed.

→ If body statements are executed only when the condition is resulting in true. If condition is resulting in false, the else body statements will be executed. Specifying the else body is not compulsory.

→ Both if statements & else body statements will never execute. Because the condition will always result in either true or false.

Syntax &:

if (condition)

{

 Stmt 1;

 Stmt 2;

}

else

{

 Stmt 3;

 Stmt 4;

}

```
class Program1
{
    public static void main (String [] args)
    {
        int num = 8;
        if (num > 7)
        {
            SOP ("Number is above 7");
        }
    }
}
```

O/P

Number is above 7.

```
class Program2
{
    public static void main (String [] args)
    {
        SOP ("Program Started");
        int num = 4;
        if (num > 7)
        {
            SOP ("Number is above 7");
        }
        SOP ("Program Ended");
    }
}
```

O/P:

Program Started

Program Ended.

It comes out of the loop
since condition is not
satisfied.

Class Program 3

```
{  
    PSVM (String [] args)  
    {  
        int num = 12;  
        if (num < 7)  
        {  
            SOP ("Number is above 7");  
        }  
        else  
        {  
            SOP ("Number is below 7");  
        }  
    }  
}
```

O/P

Number is below 7

Class Program 4

```
{  
    PSVM (String [] args)  
    {  
        int num = 9;  
        if (num < 7)  
        {  
            SOP ("Number is above 7");  
        }  
        else if (num == 7)  
        {  
            SOP ("Number is equal to 7")  
        }  
    }  
}
```

O/P

1) Number is above 7.

2) Number is below 7

3) Number is equal to 7.

```

        else
    {
        SOP ("Number is below 7")
    }
}
}

```

16th Aug 2017

Write a JAVA program to implement a withdraw process in bank account. If balance is sufficient the code should allow to withdraw money. If the balance is not sufficient the code should not allow to withdraw money.

```

class Program
{
    PSVm (String [] args)
    {
        SOP ("Program Started");
        double accbal = 500.00;
        double amt = 300.00;
        if (amt < accbal)
        {
            accbal = accbal - amt;
            SOP ("Withdraw success");
        }
        else
        {
            SOP ("Withdraw failed");
            SOP ("Insufficient balance, Try Later");
        }
        SOP ("Account balance" + accbal);
        SOP ("Program ended");
    }
}

```

O/P:

Program Started

Withdraw Success

Account balance = 200

Program ended

** SWITCH STATEMENT

evaluate
switch (expression)

```
{
    case value1 : Stmt1;
    Stmt2;
    break;

    Case value2 : Stmt3;
    Stmt4;
    break;

    case value3 : Stmt5;
    Stmt6;
    break;

    default : Stmt7;
    Stmt8;
}
```

Class Program6

```
{
    PSVM (String [] args)
    {
        char grade = 'B';
        switch (grade)
        {
            case 'A' : SOP ("FCD");
            break;

            case 'B' : SOP ("FC");
            break;

            case 'C' : SOP ("SC");
            break;

            case 'D' : SOP ("JP");
            break;

            case 'E' : SOP ("GetLost");
        }
    }
}
```

O/P

SOP : FC

```
        default : SOP ("Invalid Grade");  
    } } }
```

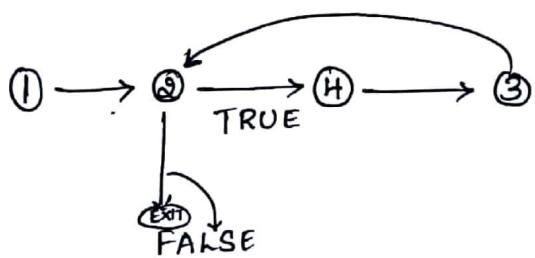
--- ** LOOP STATEMENTS

* for statement

→ to repeat the execution of statements.

** Syntax

Flow diagram :



for (initialization ; condition
 (3) inc/dec)
{
 stmt 1;
 stmt 2;
}
} → Body of
 the loop

Ex: for ($i=1$; $i \leq 5$; $i++$)

```
{ SOP ("I love JAVA"); }
```

This statement is executed
5 times.

class Program7

```
{  
    psvm (String [] args)  
    {  
        for (int i = 1; i <= 5; i++)  
        {  
            SOP(i); SOP('*');  
            SOP('JAVA');  
        }  
    }  
}
```

O/P:

1	*	JAVA
2	*	JAVA
3	*	JAVA
4	*	JAVA
5	*	JAVA

Class Programs

```
{  
    public static void main (String [] args)  
{  
        SOP( " Pgm started");  
        for (int i=1; i<=5; i++) . . . . . O/P:  
        {  
            SORint(i);  
        }  
        SOP();  
        SOP( "Pgm ended");  
    }  
-----
```

Pgm started

12345

Pgm ended

Write a JAVA program to calculate square of a number 1 to 10;

Class Program 9

```
{  
    public static void main (String [] args)  
    {  
        int i=1;  
        for (i=1; i<=10 ; i++)  
        {  
            int sqa = i*i ;  
        }  
        SOP ( "Square of "+i+" is " + sqa);  
    }  
}
```

Q) List all the even numbers between 1 to 100

```
public class Program10
{
    PSVM (String [] args)
    {
        int num;
        for (num=1; num<=100; num++)
        {
            if ((num%2)==0)           → if (num%2 != 0)
                SOP ("Even numbers are "+num);
            SOP ("Odd numbers"+num);
        }
    }
}
```

class Program11

```
{           O/P
    PSVM (String [] args)           5
    {                               4
        for (int i=5; i>=1; i++)
        {
            SOP (i);
        }
    }
}
```

** NESTED FOR Loop:

```
class Program12
{
```

17th Aug 2017

```

{
    for (int k=1; k<=5; k++)
        {
            for (int i=1; i<=5; i++)
                {
                    System.out.print(i);
                }
            SOP();
        }
}

```

outer for loop inner for loop Nested loop;

No. of Rows No. of columns.

O/P

1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5.

```
for (int k=1; k<=N; k++)
```

```
{
```

```
stmt 1;
```

→ stmt 1, stmt 2 are inner

```
stmt 2;
```

for loop runs N times.

```
for (int i=1; i<=M; i++)
```

```
{
```

```
stmt 3;
```

→ stmt 3 & stmt 4 runs

```
stmt 4;
```

N * M times.

```
}
```

Row column.

Explanation:

K = 1

K = 1

i = 1 2 3 4 5 6

Exit the loop

O/P

1 2 3 4 5

K = 5

K = 2

i = 1 2 3 4 5 6

1 2 3 4 5

i = 1 2 3 4 5 6

1 2 3 4 5

K = 6 ← Exit the loop.

K = 3

i = 1 2 3 4 5 6

1 2 3 4 5

----- K = 4

K = 4

i = 1 2 3 4 5 6

1 2 3 4 5

```

public class Program13
{
    public static void main (String [] args)
    {
        for ( K=1 ; K≤5 ; K++)
        {
            for ( int i=5 ; i≥1 ; i--)
            {
                SOP(i)
            }
            SOP();
        }
    }
}

```

O/P:

54321
54321
54321
54321

```

public class Program 14
{
    public static void main (String [] args)
    {
        for (int K=1 ; K<=5 ; K++)
        {
            for ( int i=1 ; i<=5 ; i++)
            {
                SOP(K);
            }
            SOP();
        }
    }
}

```

O/P

11111
22222
33333
44444
55555

```

    {
        class Program15
    {
        PSVM (String [] args)
        {
            for (int K=5; K≥1; K--)
            {
                for (int i=1; i≤5; i++)
                {
                    System.out.print (K);
                }
            } } SOP();
        }
    
```

O/P
5 5 5 5 5
4 4 4 4 4
3 3 3 3 3
2 2 2 2 2
1 1 1 1 1

```

class Program16
{
    PSVM (String [] args)
    {
        for (int K=1; K≤5; K++);
        {
            for (int i=1; i≤K; i++)
            {
                System.out.print (i);
            } SOP();
        }
    }
}

```

O/P
~~1 2 3 4 5~~
R₁ 1
R₂ 1 2
R₃ 1 2 3
R₄ 1 2 3 4
R₅ 1 2 3 4 5
C₁ C₂ C₃ C₄ C₅

→ It should print till
Column = Row.

∴ R₅ ≤ C₅
→ when it is n, it will
print till R_n C_n

K=1	1
K=2	1 2
K=3	1 2 3
K=4	1 2 3 4
K=5	1 2 3 4 5

```

class Program17

```

```

        SOP (K);

```

O/P:
1
2 2
3 3 3
4 4 4 4

```

for (int K=5; K≥1; K--)
{
    for (int i=1; i≤K; i++)
    {
        cout (i);
    }
}

```

K=5 i=1 1 ≤ 5 1 2 3 4 5
 2 ≤ 5
 3 ≤ 5
 4 ≤ 5
 5 ≤ 5

Row → Increment
 Column → Decrement

NOTE:

```
for( ; x < 5; x++)
{
    sop(x);
}
```

This works only if x is declared as global variable.

$$\textcircled{2} \quad \operatorname{int} x = 1;$$

$\text{for}(\ ; x \leq 5;)$

$$\{ \text{sop}(x) \}$$

Since

↓

increment
is not
provided

Go stop printing only,
just put $x++$ after $SOP(x)$:

$$\textcircled{3} \quad \inf x = 1$$

```

for( ; ; )
{
    sop(x);
}

```

→ This prints 1 00.

→ If we put $x++$ also it prints 1 00.

→ To stop this we need to use keyword

* WHILE Loop:

- while loop runs until the condition becomes false.
- While loop is faster than the for loop.
- The body of while loop should contain the statement that makes the condition false.

Example:

```
class program18
{
    PSVM (String [] args)
    {
        SOP ("Pgm started");
        int i=1;
        while (i<=5)
        {
            SOP(i);
        }
    }
}
```

→ This prints i value ie 1 only.

```
class Program19
```

```
{
    PSVM (String [] args)
    {
        SOP ("Pgm started");
        int i=1;
        while (i<=5)
```

```
    {
        SOP(i);
        i++;
    }
}
```

→ This prints
1 2 3 4 5

When we know exact no. of times to print the statement
use for loop. else use while loop.

Ex: Withdraw money 10 times → use for loop.

Withdraw money till balance becomes zero → use while loop.

** FUNCTIONS OR METHODS

→ to perform any task or operation.

* Reusability → work once/run multiple.

* modularity → splits big tasks into smaller tasks.

** Syntax:

access Specifiers Modifier returntype functionname (Arguments)
{
 Stmt1;
 Stmt2;
 Stmt3;
 return value;
}

} Body of the function.

18th AUG 2017

collection of functions →
Library

** TYPES OF FUNCTIONS:

* Built-in functions: functions provided by the language.

* User-defined functions: functions defined by programmer.

** JAVA does not support local functions. It should define in the class body. ^{It can be defined before a main method or after main method).} **

- When User defined functions are created, it should be called by call statements, else it will not be executed.
- JVM executes the program from the main method. If the user defined function is called in the main function, then JVM goes to the User defined function & executes the user defined functions. **

Examples:

Class Program1

```
{  
    // User defined functions  
    static void function()  
    {  
        SOP ("Running function()");  
    }  
}
```

// main function.

PSVM (String [] args)

```
{  
    SOP ("Pgm started");  
}
```

function(); // function calling or function invocation;

function(); // _____;

```
} } SOP ("Pgm ended");
```

O/P:-

Pgm started

Running function()

Running function()

Pgm ended.

** Functions:

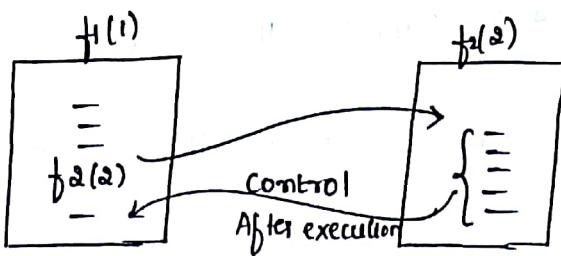
- It is a block of code defined to perform operation or task.
 - The function has two parts : 1) Function declaration.
 - 2) Function defn or body.
 - The Function declaration specifies access specifiers, modifiers, function outtype, function name and function arguments.
 - The Function defn defines the task of the function.
 - In JAVA language, the fun's are declared inside the class body. We cannot declare a function in another function body.
 - The JAVA begins the execution by calling the main function. Hence, the main function is also known as entry point of execution.
 - JVM will not run the user defined fun's or other built in function until we call the functns.
 - A function can be called from execution by using function calling statements which is name of functions.
- * A JAVA program can be written without main method.
- compiler will not throw an error but
 - @ the time of execution it throws an error (Run-time error).

Class Programs

```
{  
    // user defined functions  
    static void function1()  
{  
        SOP ("started function1() execution....");  
        function1();  
        SOP ("ended function1() execution....");  
    }  
  
    // User defined function.  
    static void function2()  
{  
        SOP ("started function2() execution....");  
        SOP ("ended function2() execution....");  
    }  
  
    // main method.  
    public static void main (String [] args)  
    {  
        SOP ("main method started");  
        function1();  
        SOP ("main method ended");  
    }  
}
```

O/P

main method started
started junction1() execution....
started junction2()
ended junction1()
ended junction2()
main method ended.



→ called functns
(functions which call other funs.)

class Program3

```
{ static void testFunction (int arg1)
```

only declaration
must be done in
argument no
initialization.

```
{
    SOP (" running value is "+arg1);
    SOP (" running testFunction...");
```

Initialization must
be done in main
method
or pass the
value

```
psvm (String [] args)
```

O/P

main method started

running testFunction

running value is 25

running testFunction

running value is 85

main method ended.

}

```
SOP (" main method started");
```

```
int x=12;
```

```
testFunction(x);
```

```
int y=12;
```

```
testFunction(y);
```

} → can be done in this
manner also.

Q) Write a function to calculate the square of the given number and display it when calling the function, we must pass the number.

```
public class Program4
```

```
{  
    static void square (int i)  
    {  
        int sqrts = i*i ;  
        SOP (sqrt);  
    }  
}
```

O/P:

```
public static void main (String [] args)  
{  
    square (2);  
}  
}
```

4

Q) Write a function to calculate Simple Interest.

```
public class Program5
```

PTR
100.

```
{  
    static void interest (double p , int t , double r)  
    {  
        double res = (P*T*R)/100;  
        SOP (res);  
    }  
    psvm (String [] args)  
    {  
        interest (2000.00 , 10.00 , 5.00);  
    }  
}
```

- If a function want to work for multiple values, use Arguments . else don't use arguments.
- When we are returning the value instead of SOP , we should not use void .
- * In place of void write the datatype of value .
- When we return the value, that value should be stored in the caller function (main method). (This is if we not store value it wont print it in console . But process takes place .)

Class Program5

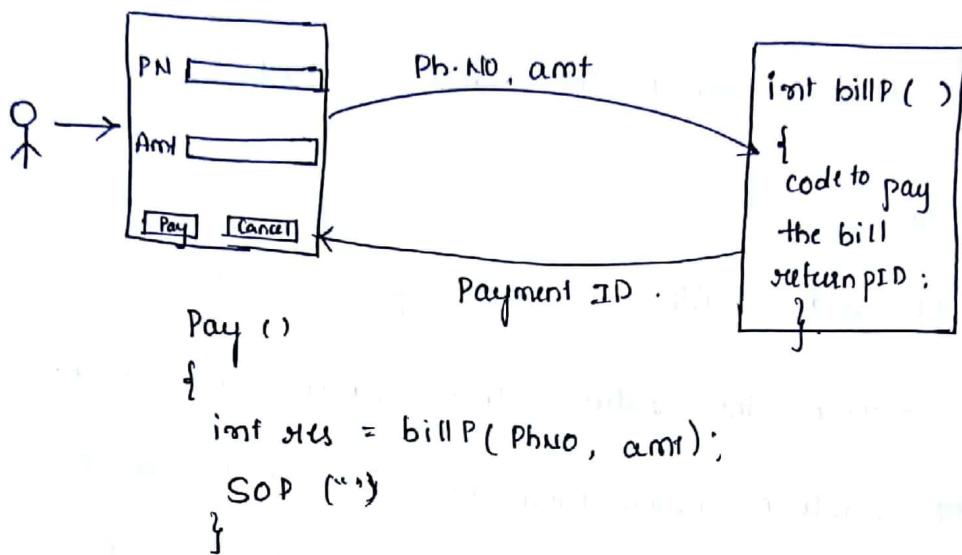
```

{ static double simpleinterest (double p, int t, double r)
  {
    double si = (p*t*r)/100;
    return si; // return value of si
  }

public static void main (String [] args)
{
  SOP ("main method started");
  double x= 10000.00;
  int y = 12;
  double z = 5.5;
  double res = simpleinterest (x, y, z); // res contains value
  // returned by the fn.
  SOP ("SimpleInterest" + res);
}
} SOP ("main method ended");
  
```

19th Aug 2017:

** Real-time example of FUNCTIONS()



void means
→ return nothing

** Area of a circle:

```
class Programs  
{  
    static double area ( double rad)  
    {  
        final double pi = 3.142;  
        double a1 = pi*rad*rad;  
        return a1;  
    }  
}
```

```
psvm (String [] args)
```

```
{  
    double r1 = 2.1;  
    double area = (r1 * r1);  
    SOP ("Radius " + r1);  
    SOP ("Area " + area);  
}  
}
```

Q) Write a program / function to withdraw money from account
if balance is sufficient withdraw should be success, else withdraw
should be failed.

```
class Programs
```

```
{  
    static void withdraw (double acc, double draw)  
    {  
        if (acc > draw)  
        {  
            account = acc - draw;  
            SOP ("Drawn")  
        }  
        else  
        {  
            SOP ("Not sufficient fund" + account)  
        }  
    }  
}
```

```
psvm (String [] args)
```

```
{  
    double x = 5,000;  
    double draw = 3,000;  
    withdraw (5,000, 3000);  
}
```

** LOCAL VARIABLES & GLOBAL VARIABLES :

Ex: 3) {

int $x = 10;$

{

$x = 30;$

int $x = 50;$

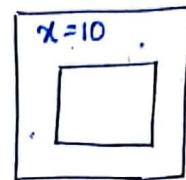
SOP(x) ; // 50 → Preference
to Local

$x = 60;$

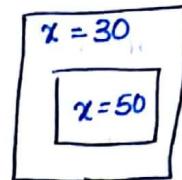
}

SOP(x) ; // 30

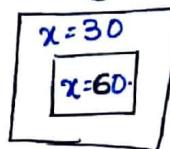
①



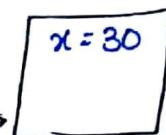
②



③



④



Variables

Exit of the loop erases the
memory of
Local variables.

Example 1: {

int $x = 10;$

SOP(x); → 10

}

SOP(x); // → Compile time error.

Ex 2:

{ int $x = 10$

{

int $y = 20;$

SOP(x); → 10

}

SOP(y); → 20

SOP(x); → 10

SOP(y); → CTE

In JAVA
No concept of
GLOBAL VARIABLES

** READING INPUTS FROM KEYBOARD:

Step 1 : import java.util.Scanner;

 └→ first statement of JAVA Code .

Step 2 : Scanner s1 = new Scanner (System.in);

 └→ Any function body .

Step 3 : Use functions to read input from Keyboard .

next () → read string value .

nextInt () → read int value .

nextDouble () → read double value .

next Long () → read long value .

Example :

```
import java.util.Scanner;
class Program1
```

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

```
{        Scanner in = new Scanner (System.in)
```

```
        String name ;
```

```
        int age ;
```

```
        System.out.print ("Enter the name") ;
```

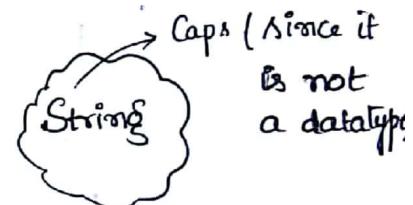
```
        name = in.next () ;
```

```
        System.out.print ("Enter the Age") ;
```

```
        age = in.nextInt () ;
```

```
        if (age > 18)
```

```
        {            System.out.println (name + " is eligible to vote ") ;
```



```

        }
    else
    {
        SOP ( name + " not eligible for vote" );
    }
}
-----
```

O/P

Enter name
Ramesh
Enter age
21
Ramesh is eligible
for vote.

```
import java.util.Scanner;
```

```
class Program
```

```
{
```

```
PSVM ( String [] args)
```

```
{
```

```
    SOP (" Pgm started" );
```

```
    Scanner S1 = new Scanner ( System.in );
```

```
    char c1;
```

```
    SOP (" type Y or N" );
```

```
    c1 = S1.next () . charAt ( 0 ) ;
```

```
    if ( c1 == 'y' ) {
```

```
        SOP (" Means YES" );
```

```
}
```

```
    else if ( c1 == 'n' ) {
```

```
{
```

```
        SOP (" Means NO" );
```

```
}
```

```
    else {
```

```
        SOP (" Invalid Input" );
```

```
        SOP (" Pgm ended" );
```

```
    }
```

** OOPS :

* JAVA type Defn : JAVA supports 4 types of Programming :

1) class type

2) interface type

3) enum type

4) annotation type .

* Syntax : 1) class Classname 2) interface InterfaceName

```
class Classname {  
    // Declaration & Definition of members  
}
```

```
3) enum enumName 4) @interface AnnotationName  
enum enumName {  
    // Declaration & Definition of members  
}  
@interface AnnotationName {  
    // Declaration & Definition of members  
}
```

* Class type Defn :

```
class classname ← declaration  
{  
    // declare & initialize variables } ← Member Variable  
    :  
    // declare & define functions } ← Member function  
}
```

Body of → a class ← Members of class

** Members of class:

2 types \rightarrow Static Members / class Members

$\&$ Non-Static Members / Instance Members.

* Variables according to Scope.

* Local Variables.

* Member Variables.

- Anything defined inside the body of a class is known as Members of a class.
- In a class body we can declare & initialize the variables, we can declare & define functions.
- The variables declared inside the body of a class is known as Member Variables.
- The functions defined inside the class body is known as Member Functions.
- Any variable declared inside the function body is known as Local Variables.
- The members of a class are classified into two types.
 - * Static Member &
 - * Non-Static Member
- A typical class defn. looks like below:

class Demo1

{
 static int x = 10; → Static member variable

 int y = 20; → Non-static member variable

 static void test1(); → Static member function

{
 int a = 20; → Local variable

}

 void test2(); → Non-static member function

{
 int b = 30; → Local variable

}

Example: class Demo1

{ void test()

{
 SOP("Running test()....");

}

class Demo2

{

 PSVM (String [] args)

{

 SOP ("Running Demo2 class");

}

class Demo3

{ PSVM (String [] args)

```

    {
        SOP ("running Demo3");
    }
}

```

3. class file will be created during/after compilation
 Demo1
 Demo2
 Demo3.
 & should execute 1 by 1.

- In a JAVA source file we can define multiple class file defns. Class file might have or may not have a main method.
- When we compile such source files, the compiler generates individual class file for each class defn. JVM can execute a class if it is having the main method. Since a main method is an entry point for execution.

```

// Functional class
class Demo1
{
    static int x=12;
    int y=34;

    static void test1()
    {
        SOP ("running test1() ....");
        int i=1;
    }

    void test2()
    {
        SOP ("running test2() ....");
    }
}

```

```

int j = 15;
}

// Main class
class Demo {
{
    public static void main (String [] args)
    {
        SOP ("Running Demo class");
    }
}

```

** ACCESSING STATIC MEMBERS OF A CLASS:

|-----|
classname. membername

// Functional class

```

class Demo {
{
    static int x=12;
    static void test1()
    {
        SOP ("running test1()....");
    }
}

```

O/P

```

main method started
12
running test1()
main method ended.

```

// Main class

```

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

```

SOP ("main method started")

SOP (Demo.x);

Demo.test1();

SOP ("main method ended")

}

}

II Main class

```
class Mainclass
```

۹

PSvm (String [] args)

{

SOP (" main method started");

SOP (Demor. x).

Demotest;

Demand → Re-initialize

`Demol.x = 25; // Assignment - can be done for static variables`
`SOP (Demol.x);`

SOP ("main method ended");

O/P:

main method started

12

Demolition

25

main method ended

}

** ACCESSING OF NON-STATIC MEMBERS :

22nd Aug 2017

`new classname(). membername.`



Object

Not Operator

Not operator (name of variable or
function name)

→ Create object of class OR instance of class.

→ new operator is used to create objects.

Syntax of Object Creation

```
new classname();
```

Example:

```
class Demo2
{
    int x = 12;

    void test()
    {
        SOP("running test() method ....");
    }
}
```

O/P:

12

running test()
method

class Mainclass

```
{
    PSVM (String [] args)
    {
        SOP ( new Demo2().x ); // variable access of Non-static
                               member
        new Demo2().test(); // method access of Non-static
                           member.
    }
}
```

class Demo2

{

int x=12;

void test()

{

}

new Demo2()

↓

Allocate memory

- Load copy of class

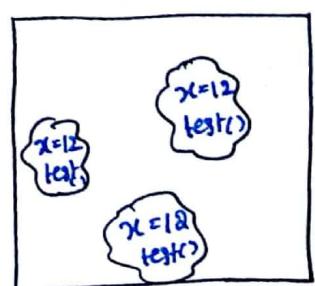
new Demo2();

=

new Demo2();

=

→ We can have n objects.



- Any members of the class declared with the static keyword are known as Static members of the class.
- Static members are associated with class & it must be accessed by class name & . operator.
- Static members of the class are loaded one copy per class in the memory.
- Any member declared without static keyword are known as Non-static members.
- The Non-static members are associated to the object of the class, hence it can be accessed by creating the object.
- The Non-static members are loaded one copy per object. We can create multiple objects of a class. Hence we can have multiple copy of the non-static members.
- The Non-static members are also known as Instance Members.

** VARIABLES : According to type of Declaration

Primitive Variables :

- are declared using datatype.
- are used to store value/data.

Ex: int x;
double y;

* Non-primitive Variables: or User defined Variables:

→ are used to store address.

→ are declared using any 3 java types * class type

* interface type

Ex: Pen p1; Car c1;

* enum type

Book b1; Bus b1;

Table t1; Mobile M1;

→ Also called as "Reference Variables"

** NEW OPERATOR:

→ new is a operator keyword provided by JAVA language to create an object or a instance of a class.

→ The new operator reserves a memory location in memory, then loads copy of a class in the memory location.

while loading the copy it loads only the non-static members of a class.

→ After creating a Object, the new operator returns the address of memory, where the object is created.

* Declare Non-primitive Variable / Reference Variable.

{classname Variablename;} }

Ex: Demo d1;

Initialize Reference Variables:

1) Initialize to null

d1 = null;

or Initialize to object of class

d1 = new Demo2();

We cannot use d1, if use JVM
throws Exception.

→ we access object using d1.

Example: class Demo2

```
{  
    int x=12;  
    void test()  
    {  
        SOP("Running test() method ...");  
    }  
}
```

class MainDemo

```
{  
    PSVM (String [] args)  
    {  
        SOP ("Pgm Started");  
        Demo2 d1; // Declaration of object.  
        d1 = new Demo2();  
  
        SOP (d1);  
        SOP (d1.x);  
        d1.test();  
    }  
}
```

```

Ex2: class Demo3
{
    int x = 12;
    double y = 34.56;
}

```

```
class Mainclass3
```

```
{
    PSVM (String [] args)
}
```

```
{
    SOP ("Pgm started");
}
```

```
Demo3 d1 = new Demo3();
```

```
Demo3 d2 = new Demo3();
```

```
SOP ("d1 ---> " + d1);
```

```
SOP ("d2 ---> " + d2);
```

```
--> SOP ("x value of 1st object: " + d1.x);
```

```
SOP ("y value of 1st object: " + d1.y);
```

```
SOP ("x value of 2nd object: " + d2.x);
```

```
SOP ("y value of 2nd object: " + d2.y);
```

```
SOP ("Pgm ended");
```

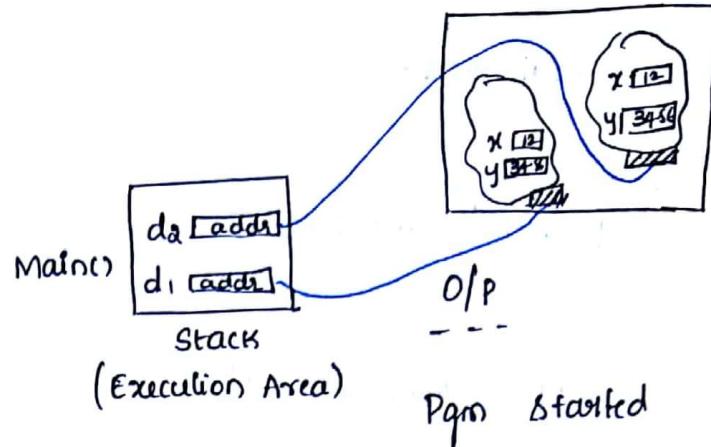
```
}
```

```
--> d1.x = 45;
```

```
d1.y = 7.2;
```

x value of 2nd object: 12

y value of 2nd object: 34.56



$d_1 \rightarrow \text{Demo3@1db9742}$

$d_2 \rightarrow \text{Demo3@106d69c}$

x value of 1st object: 12

y value of 1st object: 34.56

x value of 2nd object: 12

y value of 2nd object: 34.56

Pgm ended.

O/P:

Pgm Started

$d_1 \rightarrow$

$d_2 \rightarrow$

x value of 1st object: 45

y value of 1st object: 7.2

only
change in
 d_1 value &
hence there is
no change in

class MainClass

{

 psvm (String [] args)

{

 SOP ("Pgm Started");

 Demo3 d1 = new Demo3();

 Demo3 d2 = d1 // copy value of d1 to d2.

 SOP ("d1 → " + d1);

O/P

 SOP ("d2 → " + d2);

Pgm started

 d1.x = 45;

d1 →

 d1.y = 7.2;

d2 →

 SOP (d1.x); SOP(d1.y); SOP(d2.x); SOP(d2.y);

45

7.2

45, 7.2

 SOP ("Pgm ended");

}

}

Ex: Google Drive

If anything is
changed by a
single user, it
reflects to others
also.

23rd Aug 2017

** REFERENCE VARIABLES

→ Any variable declared using class type is known as Reference Variable.

→ For Reference variables we can initialize either null or Object.

→ If the reference variable is initialized to null, we cannot use

such reference variables to access. If we try to use it throws NullPointer Exception () .

→ For a Reference variable we can initialize using that reference variable with object ; we can access the properties of the object .

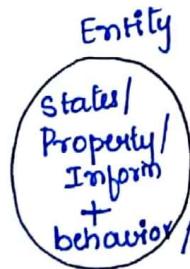
→ Reference Variable holds address of the object . If we don't have reference to the object , then we cannot access the object property .

→ If we modify the properties of one object , it will not reflect in another object .

→ An object can be pointed by more than one Reference Variable .

→ changes made to the object through one Reference will reflect in the another ^{other} Reference also .

** OBJECT & CLASS



class NoteBook
{
 int pages = 50 ;
 double price = 25.00 ;
 void open ()

Ex: Marker Pen

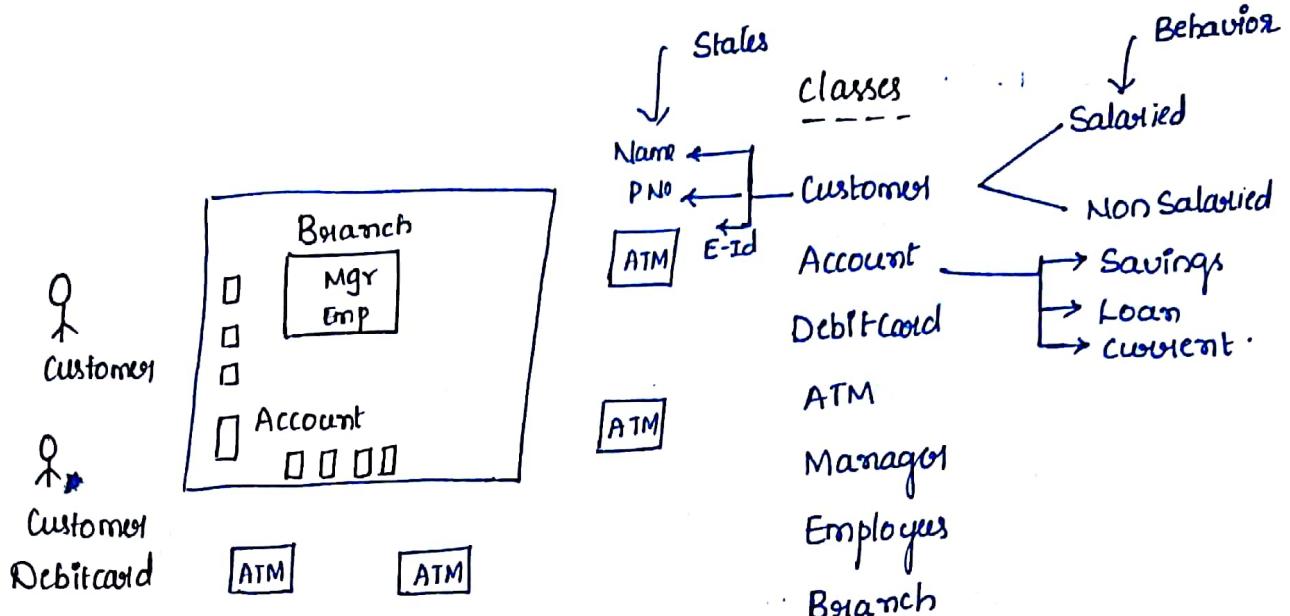
↓
color
brand
price
shape
weight
Write ()
⋮

NoteBook

↓
pages open()
Size close()
price least()
brand rotate()
color
weight

- Any entity having its own state and behavior is known as OBJECT.
- The state represents information of the object
The behaviour represents functionality of the object
- A Class is a definition block used to define the state and the behaviour of the object.
- The Non-static members ^{Variable} of a class defines the state of the object; whereas the Non-static members ^{fun} defines functionality of the object.
- From a class defn., we can create multiple objects by using new operator.
- To refer each object, we use Reference Variables.
- CLASS is a logical entity whereas
OBJECT is a physical entity because object is stored in the memory.
- We can create multiple objects from a one class definitions. These objects will have same property but might differ in values of the property.

Example : Banking System.



** With one class many objects can be created

class Circle

{

double rad;

final static double pie = 3.14;

void diameter()

{

double dia = 2 * rad;

SOP ("Diameter is " + dia);

}

void area()

{

double area1 = pie * rad * rad;

SOP ("Area is " + area1);

}

void circumference()

{

double c1 = 2 * pie * rad;

SOP ("Circumference is " + c1);

}

```
class MainClass4
```

```
{
```

```
PSVM (String [] args)
```

```
{
```

```
SOP ("Main method is started");
```

```
Circle c1 = new Circle();
```

```
c1.radius = 2.1;
```

O/P

```
SOP ("Radius is " + c1.radius);
```

Main method is started

```
c1.diameter();
```

Radius is 2.1

```
c1.circumference();
```

Diameter is 4.2

```
c1.area();
```

Circumference is

```
Circle c2 = new Circle();
```

Area is

```
c2.radius = 3.1;
```

Radius is 3.1

```
SOP ("Radius is " + c2.radius);
```

Diameter is 6.2

```
c2.diameter();
```

Circumference is

```
c2.area();
```

Area is

```
c2.circumference();
```

```
SOP ("Main method is ended");
```

```
}
```

Q) Write a program to display Area of a rectangle.

```
class Square
```

```
{
```

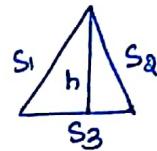
```
double length; breadth;
```

```

void area ()
{
    double areal = side * side;
    SOP ("Area is " + areal);
}

void perimeter ()
{
    double perim = 4 * side;
    SOP ("Perimeter is " + perim);
}

```



class Mainclass

```

{
    PSVM (String [] args)
    {
        Square Sq1 = new Square ();
        Sq1.length = 4.0;
        Sq1.area ();
        Sq1.perimeter ();
    }
}

```

** ARRAYS :

→ To store multiple values of same type.

Array Declaration:

$\boxed{\text{arraytype} [] \text{ arrayname};}$

24th Aug 2017

OR

arraytype arrayname [] ;

** ARRAY INITIALIZATION:

[arrayname = new arraytype [n];]

n → no. of elements or array size.

JVM allocates 'n' continuous memory.

Ex: int [] a; a; → a; is the array of int type.

a; = new int [5];

↳ Create array of 5 memory locations configured to store int values.

1st and 3rd 4th 5th element
a; [0 | 0 | 0 | 0 | 0] → Default values.
0 | 1 | 2 | 3 | 4 → Index.

n → Size of Array.

Index Range is 0 to n-1.

Print and element : SOP (a[i]);

5th element : SOP (a[4]);

Index of array starts from 0 bcoz it follows the Number System.
Ex: Hexadecimal 0 to 15;

Example 1:

```
class Mainclass  
{  
    psvm (string [] args)  
    {  
        SOP ("Pgm started");  
        int a1 = new int [5];
```

//Store values in array

```
a1[0] = 12 ;
```

```
a1[1] = 41 ;
```

```
a1[2] = 65 ;
```

```
a1[3] = 51 ;
```

```
a1[4] = 75 ;
```

O/P:

0 → 12

1 → 41

2 → 65

3 → 51

4 → 75

```
SOP ("array elements are");
```

```
for (int i=0; i<=4; i++)
```

```
{  
    SOP (i + " ---> " + a1[i]);
```

```
}
```

```
SOP ("Pgm ended");
```

```
}
```

Q) Write

```
a program to print only even positions
```

```
---> if (i%2 == 0)
```

```
SOP (i + " --->
```

12	41	65	51	75
----	----	----	----	----

O/P:

0 → 12

2 → 65

4 → 51

Q) To print even numbers of an array.

```
for (int i=0; i<=4; i++)  
{  
    if (a1[i] % 2 == 0)  
        SOP (i + "---->" + a1[i]);  
}
```

O/P
0 → 12

Example :-

```
class Mainclass2  
{  
    PSVM (String [] args)  
    {  
        SOP ("Pgm started");  
  
        int [] a1 = new int [1, 2, 3, 4, 5]; // Array initializer - 2nd method  
        SOP ("Array elements are");  
  
        for (int i=0; i<=4; i++)  
        {  
            SOP (i + "---->" + a1[i]);  
        }  
        SOP ("Pgm ended");  
    }  
}
```

Q) Write a java pgm to read a no from a keyboard, if no is found in array display msg no found in array else no not found in array.

```
class Mainclass3
```

```
{
```

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

```
{
```

```
    Scanner s1 = new Scanner (System.in);
```

```
    int arr [] = new arr [5];
```

```
    arr [0] = 2;
```

```
    arr [1] = 4;
```

```
    arr [2] = 6;
```

```
    arr [3] = 8;
```

```
    arr [4] = 10;
```

Or int arr [] = { 2, 4, 6, 8, 10 }

```
SOP (" Enter a number to be found");
```

boolean numFound = false;

```
int num = s1.nextInt();
```

for (int i=0; i<=4; i++)

```
for (int i=0; i<=4; i++)
```

{
if (arr[i] == num)

```
{  
    if (arr[i] == num)
```

{
 numFound = true;

```
}  
    SOP (" No. found in an array");
```

break;

}

if (numFound == true)

{
 SOP (" No. found");

```
else
```

```
{  
    SOP (" No. not found in an array");
```

else

{

SOP (" No. not found");

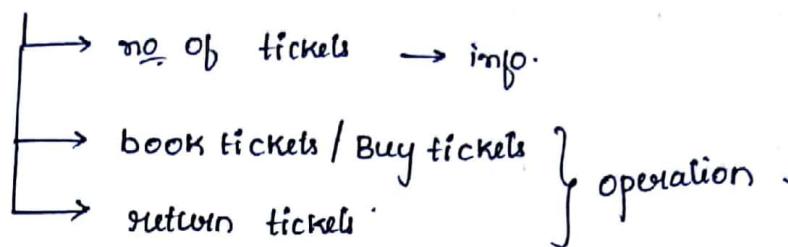
```
}  
}
```

Q) Write a program to find a number of occurrences of a given number. The number should be read from Keyboard.

```
Scanner s1 = new Scanner(system.in);
int [] a1 = { 2, 5, 2, 8, 2 } // array initializes
int count = 0;
SOP (" Enter a no to be found ");
int num = s1.nextInt();
for (int i=0; i<=4; i++)
{
    if (a1[i] == num)
    {
        count++;
    }
}
if (count > 0)
{
    SOP ( count + "times");
}
else
    SOP (" No found");
}
```

29 AUG 2017

Ticket Counter



Class TicketCounter

```
{  
    int no_tickets = 50;  
  
    void bookTickets (int n)  
    {  
        SOP ("Booking " + n + " tickets");  
        no_tickets = no_tickets - n;  
    }  
  
    void returnTickets (int n)  
    {  
        SOP ("Returning " + n + " tickets");  
        no_tickets = no_tickets + n;  
    }  
  
    void availableTickets ()  
    {  
        SOP ("Total tickets available: " + no_tickets);  
    }  
}
```

Proper way

```
void bookTickets (int n)  
{  
    SOP ("Booking " + n + " tickets");  
    if (n < no_tickets)  
    {  
        no_tickets = no_tickets - n;  
        SOP ("Booking Success");  
    }  
    else  
    {  
        SOP ("Booking tickets failed");  
    }  
}
```

Class Mainclass

{

PSVM (String [] args)

{

SOP ("pgm. started");

TicketCounter tc1 = new TicketCounter();

tc1. availableTickets();

tc1. bookTickets (5);

tc1. bookTickets (5);

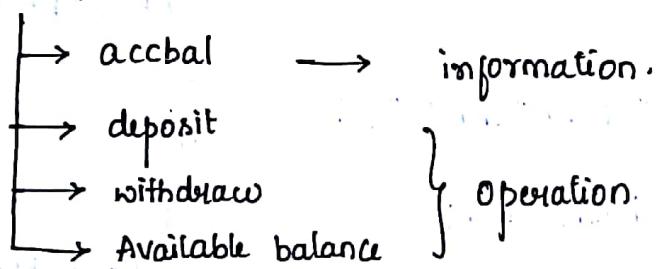
tc1. availableTickets();

SOP ("pgm. ended");

}

}

** BANK ACCOUNT



Class BankAccount

{

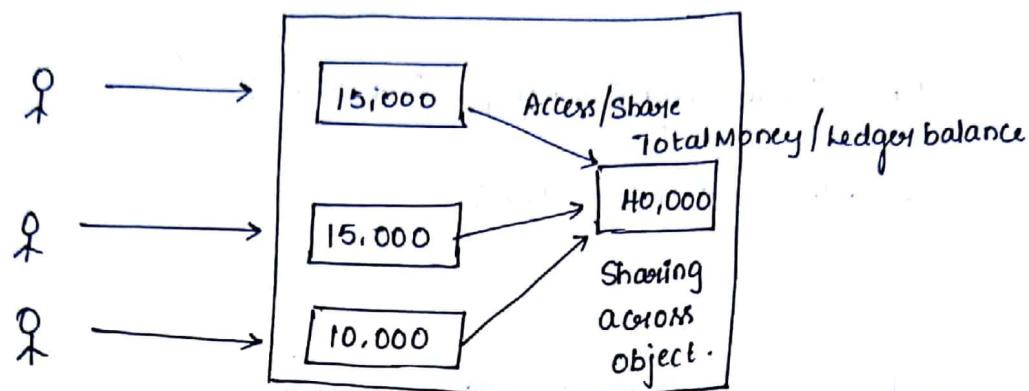
double accbal = 30,000.00;

```
void class deposit ( double cash )  
{  
    SOP ( "The deposit money is" + cash );  
    accbal = accbal + cash ;  
}
```

```
void class withdraw ( double cashes )  
{  
    SOP ( "The money to be withdraw" + cashes )  
    if ( accbal > cashes )  
    {  
        SOP ( "Money withdraw is successfull" + cashes );  
        accbal = accbal - cashes ;  
    } else  
    {  
        SOP ( "Sufficient fund not found" + accbal );  
    }  
}
```

```
void class Available balance ()  
{  
    SOP ( "Available balance" + accbal );  
}
```

** When should we use STATIC?



Same Program as Previous.

```
void withdraw( double amt )
```

```
{
```

```
SOP ( "Withdrawing Rs" + amt );
```

```
if (amt <
```

```
class Bank
{
    static double ledgerbalance = 0.0;
    static void totalmoneyInBank()
    {
        SOP (" Total money : " + ledgerbalance);
    }
}
```

class Mainclass6

```
{
    PSVM (String [] args)
    {
        SOP (" main method started");
        Bank::totalMoneyInBank();
        Account a1 = new Account();
        Account a2 = new Account();
        Account a3 = new Account();
        a1.deposit (10000.00);
        a2.deposit (5000.00);
        a3.deposit (15000.00);
        Bank::totalmoneyInBank ();
        a1.deposit (5000.00);
        a1.availableBalance ();
    }
}
```

```

Bank::totalMoneyInBank();
a2::deposit(10000.00);
a2::availableBalance();

Bank::totalMoneyInBank();
a3::withdraw(5000.00);
a3::availableBalance();

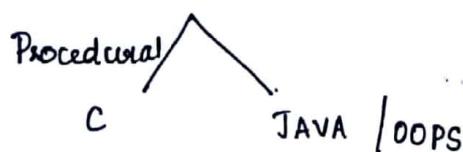
Bank::totalMoneyInBank();
SOP("main method ended");
}
}

```

- STATIC MEMBERS are used whenever we have to share the data or function across the object.
- If data as well as function is specific to the object or particular to the object then we go for Non-static function.

⇒ Temperature is 24°C

⇒ Price is 2500



```

void
{
    - - - ++
    - - - -
}

```

→ This statement is incomplete

bcoz Object is not defined

for which the temp is 24°C

Correct one: → object.
AC temp is 24°C.

30 JANG 2017

** INITIALIZATION OF MEMBER VARIABLES

- A local variables must be initialized before using it in any operation otherwise compiler throws an error.
- The member variables of a class can be used without initialization. if code does not initialize member variables, compiler provides default initialization.
- All integer type variables will have zero as default value, float type, value are initialized to 0.0, char type initialized to ' \u0000' & boolean variable type is false.

Examples: class Demo1

```
{  
    int p;  
    boolean q;  
    float r;  
}  
Demo1 d1 = new Demo1();  
SOP(d1.p); → 0  
SOP(d1.q); → false  
SOP(d1.r); → 0.0
```

** 3 WAYS OF INITIALIZATION OF MEMBER VARIABLES:

a) at the time of declaration.

b) Blocks

- Static block → initialize Static member variables
- Non-static block → Non-static member variables

c) Constructors.

Flow of Execution:

- 1) Loading time initialization
- 2) Blocks

- 3) Constructors.

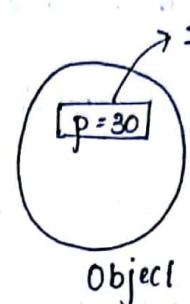
Class Demo1

```
{  
    int p = 10;  
}
```

```
{  
    p = 20;  
}
```

Demo1()

```
{  
    p = 30;  
}
```



** CONSTRUCTORS :

→ Members of class used to initialize the states of the object
@ Object creation time

↓
(property)

→ Syntax :

↖ Access > NOT compulsory
ConstructorName (<arguments>)

```
{  
    // code to do initialization  
}
```

→ RULES :

- 1) Constructor Name must be same as className .
- 2) Constructor should not specify return type .

Demo1 d1 = new Demo1();

operator
→ Allocate memory
→ copy of a class
→ call the constructor of a class

Constructor of class
↳ initialize the object.

```

class Demo1
{
    // member variables
    int p;
    double q;

    // constructors
    Demo1()
    {
        p = 12;
        q = 34.12;
    }

    // member function
    void display()
    {
        SOP ("value of p : " + p);
        SOP ("value of q : " + q);
    }
}

```

```

class Mainclass
{
    public static void main (String [] args)
    {
        SOP ("Pgm started");
        Demo d1 = new Demo1();
        d1.display();
        SOP ("Pgm ended");
    }
}

```

Parameterized Constructors

are used to use our own values randomly not fixed values.

// Parameterized Constructors

```

int Demo1 (int arg1, double arg2)
{
    p = arg1;
    q = arg2;
}

```

Demo d1 = new Demo1(10, 20);
d1.display;

Q) Write a JAVA program to represent student as an object. Define parameterized constructor to define state / property of a student.

Read the values from the keyboard.

```
class Student  
{  
    String name;  
    long phnum;  
    String emailid;
```

```
Student ( String arg1, long arg2, String arg3 )
```

```
{  
    name = arg1;  
    phnum = arg2;  
    emailid = arg3;  
}
```

```
void details ()
```

```
{  
    SOP (" " + name);  
    SOP (" " + phnum);  
    SOP (" " + emailid);  
}
```

```
}
```

```
class Mainclass3
```

```
{
```

```

public class P_SVM {
    public static void main (String [] args) {
        Scanner s1 = new Scanner (System.in);
        SOP ("enter name");
        String stname = s1.next ();
        SOP ("enter phone number");
        long stphnum = s1.nextlong ();
        SOP ("enter email id");
        String stemailid = s1.next ();
        // creating object
        Student st1 = new Student (stname, stphnum, stemailid);
        st1.display ();
    }
}

```

- CONSTRUCTORS are special members of a class, and are used to initialize the object at object creation time.
- Every class defined must have constructors either defined by the compiler or defined by user.
- Constructors defined by the compiler is known as Compiler defined Constructors and it is also known as default Constructors.

- Constructors defined by the programmer is known as User defined constructors.
- User can define 2 types of Constructors.
 - If NO arg constructors.
 - & Parameterized constructors.
- Defining a Constructors with arguments is known as Parameterized Constructor.
- Default Constructor is provided by the compiler only to the class which is not having user defined constructor.
- If class has user defined constructor compiler will not provide Default constructor.
- While defining a Constructor, a Constructor name should be same as class and constructor should not specify any return type.
- If the object must be ready for usage, then we write Constructors
- Each & every class defined in JAVA language must have ^{Constructor} Constructors but not all the classes have default, because the default constructor are available in the class which is not having user defined constructor.
- One of the use of default constructor is to provide default initialization.

CONSTRUCTOR OVERLOADING

```
class Demo3
```

```
{
```

```
    int p;
```

```
    double q;
```

```
// Overloaded constructors.
```

```
Demo3 (int arg)
```

```
{
```

```
    SOP ("running Demo3(int) constructor");
```

```
    p = arg;
```

```
}
```

```
Demo3 (double arg)
```

```
{
```

```
    SOP ("running Demo3(double) constructor");
```

```
    q = arg;
```

```
}
```

```
Demo3 (int arg1, double arg2)
```

```
{
```

```
    SOP ("running Demo3 (int, double) constructor");
```

```
    p = arg1; q = arg2;
```

```
}
```

```
void display ()
```

```
{
```

```
    SOP ("p value: " + p);
```

```
} SOP ("q value: " + q);
```

NOTE:

If we provide return type like void Demo3(), it becomes a function & it takes default values but no error.

```

class Mainclass
{
    PSVM (String [] args)
    {
        Demo3 d1 = new Demo3 (25);
        Demo3 d2 = new Demo3 (5.6);
        Demo3 d3 = new Demo3 (45, 8.2);

        d1.display ();
        d2.display ();
        d3.display ();
    }
}

```

- In a class defining more than one constructor with different argument list is known as Constructor Overloading.
- The argument list must be differ either in the type of argument or length of argument.
- Any two constructor should not match in the argument type otherwise compiler throws an error.
- If the class is having overloaded constructor, the new operator calls the relevant constructor on the basis of argument.
- If a class provides Overloaded constructor, the object of that class can be initialized based on the values needed.

→ The benefit of Overloaded constructor is the object can be initialized with different types of values.

NOTE:

→ The constructor can be overloaded but cannot be overridden.

→ Following modifiers keywords are not allowed in the constructor declaration Syntax : static, final, abstract .

→ We can set any of the four access level to the constructor.

Q) Develop a Java program to enroll a student for a course.

For enrolling the student must provide two information Name & Phone Number. Name is compulsory. Phone no is not compulsory. Student can enroll to the course without providing phone number also write interactive program for the above requirement.

class Student

{

String name;

long phnum;

Student (String arg1)

{

name = arg1 ;

}

Student (String arg1, long arg2).

{

name = arg1 ; phnum = arg2 ;

```
void details ()  
{  
    SOP (" Student name: " + name);  
    SOP (" Student phnum : " + phnum);  
}  
}
```

class Mainclass {

```
{  
    public static void main (String [] args)  
    {  
        Scanner s1 = new Scanner (System.in);  
        SOP (" Welcome to the enrolling to the course");  
        SOP (" Type 'y' if u have phone number");  
        SOP (" Type 'N' if u dont have phone number");  
        char input = s1.next () .charAt (0);  
        if (input == 'y') {  
            SOP (" Enter your name");  
            String stname = s1.next ();  
            SOP (  
                long stph = s1.nextLong ();  
                Student st1 = new Student (stname, stph);  
                SOP ("Enrollement completed");  
                st1.details ();  
            );  
        }  
    }  
}
```

```

        } else if (input == 'n') {
            SOP ("Enter the name");
            string stname = in.next();
            Student sta = new Student (stname);
            sta.details ();
            SOP ("Enrollment completed");
        }
    }
}

```

Which will be executed first?

```
class Demo1
```

```
{
    int p;
    double q;
{
    p = 10;
    q = 4.5;
}
```

```
Demo1()
```

```
int p = 12
```

```
double q = 34.12
```

O/P : 10

4.5

If p & q are
not initialized in
Demo1() then
O/P would have

p = 10

q = 4.5

p = 12

q = 34.12

We can't access these variables

because these are local variables

These cannot be accessed only it can print
only if SOP is written inside that body.

** FINAL MEMBER VARIABLES:

[01-SEP-2017]

Ex 1: class Demo1 → No default initialization.

```
{
    final int x = 10;           → Code must initialize
                                * @ declaration
    final static int y = 20;     * Blocks
                                * Constructors .
}
```

Ex 2: class demo1

```
{
    final int x;
    final static int y;
    {
        x = 10; } Non-static Block
    static { y = 20; } Static Block
}
```

Ex 3: class Demo1

```
{
    final int x;
    final static int y;
    static { y = 30; }
    Demo1()
    {
        x = 10; } Constructor
}
```

Ex 4: class Demo5

```
{
    final int x = 10;
    final static int y = 20;
    static { y = 23; }
    Demo5()
    {
        x = 45;
}
```

} Compiler throws an error.
Since final variable can
be initialized only once.

`final static` member variable → Single copy per class, cannot be initialized

`final` non-static member variable → Single copy per object, cannot be re-initialized.

class Employee

{

String name;

final int id;

Employee (int arg1, String arg2)

{

id = arg1

name = arg2;

}

void details ()

{

SOP (id)

SOP (name)

}

class Mainclass

{

PSVM (String [] args)

{

Employee e1 = new Employee (3265, "Ramu");

e1.details();

} Employee e2 = new Employee (6528, "Somu");

e2.details();

Id should be different

hence we go for parameterized

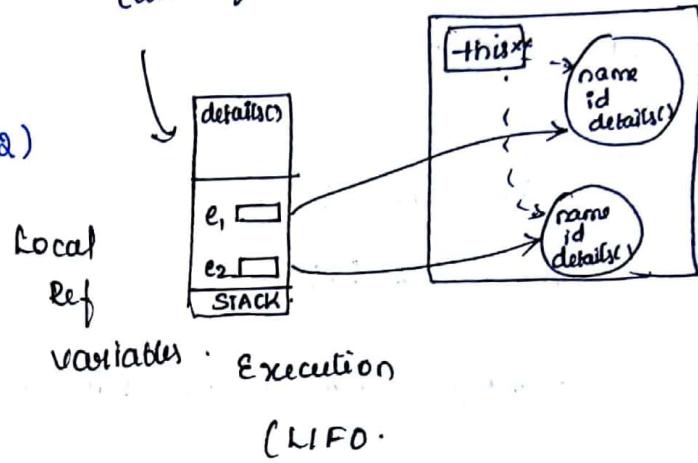
constructor

```

class Employee
{
    String name;
    final int id;
    Employee (int arg1, String arg2)
    {
        id = arg1;
        name = arg2;
    }
}

```

called functions will be always on top.



```
void details ()
```

```
{
    SOP ("this Keyword value :" + this);
    SOP ("Employee ID :" + this.id);
    SOP ("Employee Name :" + this.name);
}
```

This will be automatically prefixed by JVM

O/P:

e₁.value : Employee@1b9742

this keyword value : Employee
@1b9742

```
class Mainclass
```

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

{

```
    Employee e1 = new Employee (3265, "Ramu");
```

```
    Employee e2 = new Employee (6588, "Somu");
```

```
    SOP (e1); e1.details();
```

```
    SOP (e2); e2.details();
```

}

}

```
void details()
```

```
{  
    int id = 12;  
    String name = "Raju";  
    SOP ( this.id ); }  
    SOP ( this.name ); }  
}
```

When we try to execute this, the id becomes 12 & name becomes Raju. even when object is created. So, this requires a keyword "this" explicitly.

```
SOP (this.id);  
SOP (this.name);
```

→ JAVA language provides special keyword "this" which is used to refer current object.

→ this keyword will always point to the current object.

→ The object or references on which the function is invoked is known as current object.

→ this keyword should be used only inside the constructor or non static function body.

→ It should not be used in the static method body.

→ Whenever the member variable names are same, the member variables are differentiated by using this keyword explicitly.

Q) Why constructor name must be same as className?

It is used to know which class object is being created and load the corresponding value.

Q) Does Constructor return a value? Yes. Constructor returns the address of the object created.

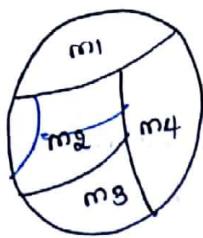
- Q) What is Constructors? why it is required?
- Q) What is Default constructor?
- Q) What is parameterized constructor?
- Q) Explain Constructor Overloading?
- Q) Does constructor return any value?
- Q) Can a class have both default & parameterized constructor.
- Q) Explain this keyword.
- Q) What do you mean by final member variable & how to initialize.
- Q) Can be constructor ^{be} static.
- Q) Does every JAVA class need a constructor. Why?
- Q) What is the difference b/w Blocks & Constructor & functions.
- Q) Develop a JAVA program to represent Account has an object.
- The account need 3 information Customername, contact number & the amount to be deposited. The customer can open the account without initial deposit or with initial deposit.

02-sep-2017

```
class Employee
{
    String name;
    int id;
    Employee (int id , String name)
    {
        this.id = id;
        this.name = name;
    }
}
```

arguments are also local variables.
where Member variables & local variables are same, make use of 'this' keyword.

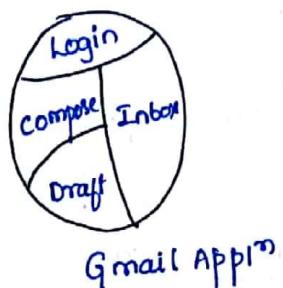
** JAVA PACKAGES:



Modules
↳ Submodules

Packages
↳ Subpackages

Ex :



Module Structure

Login
↳ Registration

Inbox
Compose
Draft

Package Structure

- JAVA package is a collection of JAVA programs and sub packages.
- A package may contain JAVA program defined by any type.
- Package having a sub-package is not compulsory.
- JAVA packages are created to achieve modularity when developing the appln.
- The JAVA package must be declared by using following syntax

`package packagename;`

** RULES OF PACKAGE DECLARATION

- * The package declaration must be first line of the source file.
- * In one source file, only one package declaration is allowed.

Ex: `package login;`

`package login.registration;`

↓
Package ↓
Subpackage

NOTE:

→ In Industry format the package name will be always in lowercase.

```
package login;  
class Demo1  
{  
}  
class Demo2  
{  
}
```

Demo1.java

```
package inbox;  
class Sample1  
{  
}  
class Sample2  
{  
}
```

Sample1.java

```
package login;  
class Run1  
{  
}  
class Run2  
{  
}
```

Run1.java

Demo1, Demo2 belongs to
Login package

Sample1, Sample2 belongs
to inbox package

Run1, Run2 belongs
to login package

** IMPORTING THE CLASS.

class name
should be differ-
ent in case
of packagename
is different

→ If a class belonging to a package wants to use the members of a class belonging to different package, then that class must be imported.

→ To import a class java provides Import Statement.

→ The Syntax of import statement is

```
import packagename.classname;
```

→ The import statement must be used after the declaration statement.

→ In one source file, we can use multiple import statement. In other words, we can import more than one class belonging to the same package or different package.

** TYPES OF IMPORT:

→ There are two types of import
 ↳ Import
 ↳ Static import.

* Import → Imports all the statements / members of the class.

* Static Import → Imports only the static members of the class.

Ex: `import java.util.Scanner;`

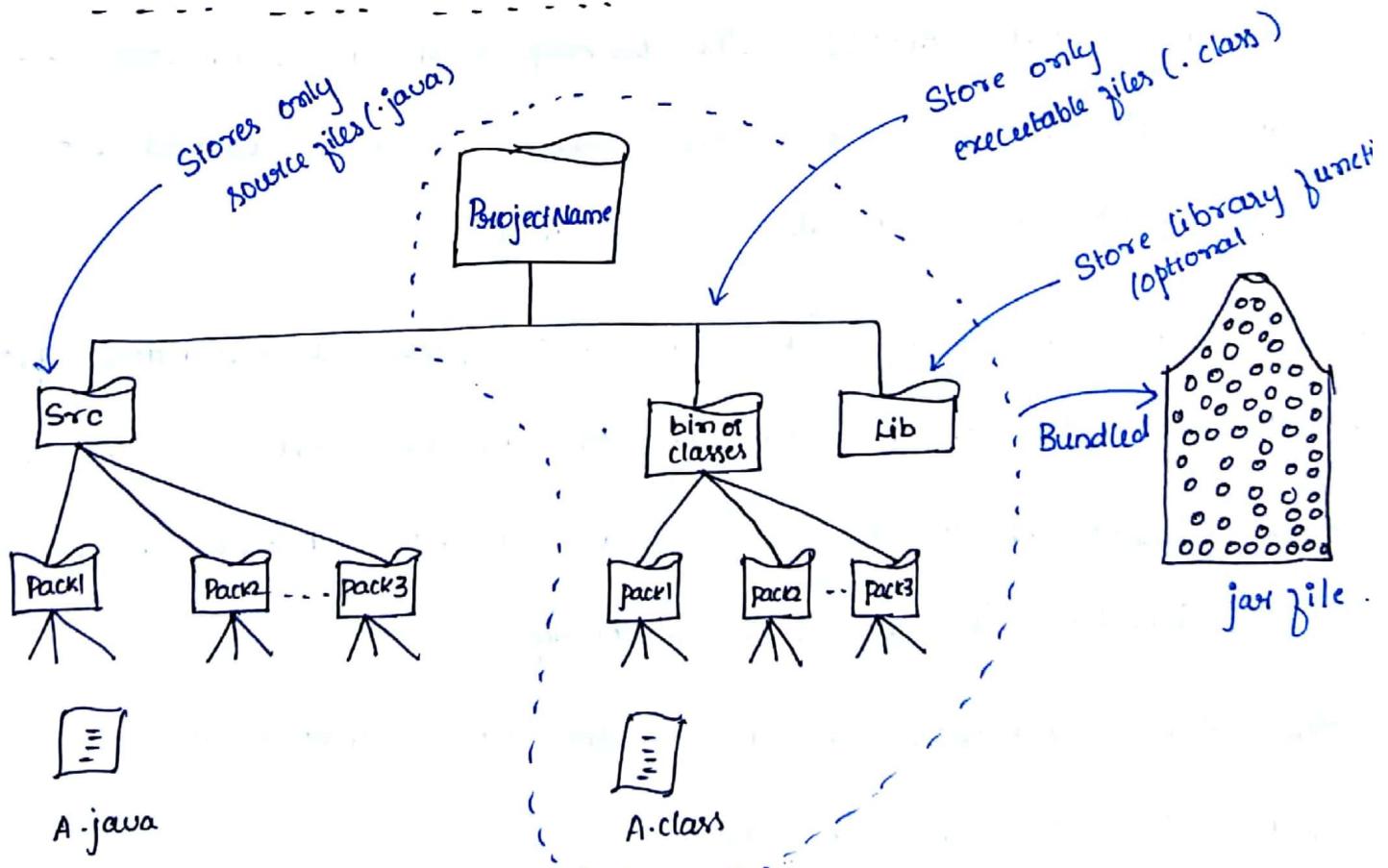
this must
be particular
class name

`import java.util.*;` → Import all the classes belonging to util package.

`import static java.lang.System.*` → Import all the static member of the System class.

Name of the package

** PROJECT FOLDER STRUCTURE:



** ACCESS SPECIFIERS

JAVA language provides 4 types of Access Specifiers.

1) Private

2) Public

3) protected

4) package Level.

→ All these access Specifiers defines the access level of the members of the class.

→ Using these access levels, we can restricts the access of members.

→ If a member is having PRIVATE ACCESS, then it is restricted only to the class where it is declared. It cannot be accessed from outside the class.

→ If we don't give any access to a class, then it will be considered as PACKAGE LEVEL ACCESS. The package level access members has a restriction upto the package. These members can be accessed from any class which belongs to the same package.

→ The members having PROTECTED ACCESS, has a restriction upto the package level. It is same as the PACKAGE LEVEL ACCESS. The Protected members of a class can be accessed from outside the package only by inheriting it (by IS-A relationship).

→ The PUBLIC MEMBERS of a class has no restrictions. It can be accessed from any package class.

Ex: To compile and Execute in Package Structure.

```
package login;  
class Demo1  
{  
    public static void main (String [] args)  
    {  
        System.out.println (" ");  
    }  
}
```

JAVA is not
purely an object
Oriented programming lang
bcz it supports
primitive variables

Ex: E:\Keshava\TECM22\demoproject\src > javac -d ..\bin
login\Demo1.java

\src > cd ..\bin

bin> java login.Demo1
O/P: ...

\bin>

OAth | Sep/2017

→ For a class definition we can give only two access at Package Level & Public

→ We cannot declare a class as private or protected:

→ Only the class which has public access can be imported to another package.

→ If class is declared as public, its members are not public,

have to declare explicitly.

- In one Src file, only one public class is allowed. If we want to write multiple public classes, then it must be defined in multiple source files. If the Src file having 'public' classname, then the filename should/must be same as classname which is having 'public' access, otherwise compiler throws error.
- The Constructors can have any of the four access. for a user defined constructor user can set any one of the access. The compiler defined constructor will always have the same access of its class.

Ex: If the class is public, then the constructor will also have public access.

** ENCAPSULATION:

- Encapsulation principle specifies that protect the members by binding the members to a class. These members can be restricted in access by using the access specifiers.
- The class ~~defn~~ encapsulates its members whereas the package encapsulates the packages, the jar file is a encapsulation of the packages.

→ In Java lang we cannot define any data without encapsulating in a class. If we try to declare a variable, outside the class, the compiler throws error.

C language

```
#include <stdio.h>
int y = 20;
```

```
void main()
```

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

```
    void incx()
```

```
{
```

```
}
```

```
}
```

This
works
(NO encapsulation)

Global Variable

Java language

```
package ...;
```

```
import ...;
```

```
class Demo1
```

```
{
```

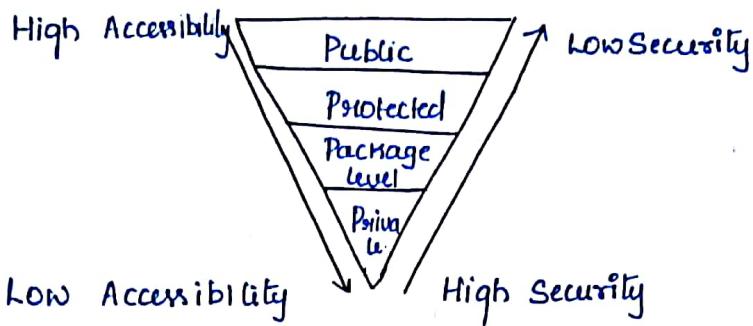
```
    int x = 10
```

```
}
```

Encapsulation

This throws
error.

ACCESS SPECIFIERS	WITHIN CLASS	WITHIN PACKAGE	FROM OUTSIDE Pkg
PRIVATE	YES	NO	NO
PACKAGE LEVEL	YES	YES	NO
PROTECTED	YES	YES	YES (IS-A relationship)
PUBLIC	YES	YES	YES



5th Sep. 2017

** JAVA BEAN CLASS

Bean \longleftrightarrow Information

- Rules:
- 1 \rightarrow Class must be public
 - 2 \rightarrow Member variables (Beans) must be private
 - 3 \rightarrow Public constructor.
 - 4 \rightarrow
 - a) Public method to get bean value \rightarrow getters
 - b) Public method to set bean value \rightarrow setters

Example:

```

① public class Employee
{
    ② private int id;

    ③ public constructor (int id)
    {
        this.id = id;
    }

    ④ a. public int getId ()
    {
        return id;
    }

    ④ b. public void setId (int id)
    {
        this.id = id;
    }
}

```

Ex:

```
package jsp.office;  
public class Employee  
{  
    private int id;  
    private String name;  
    private double salary;  
    public Employee (int id, String name, double salary)  
    {  
        this.id = id;  
        this.name = name;  
        this.salary = salary;  
    }
```

```
    public int getId()
```

```
    {  
        return id;  
    }
```

```
    public void setId(int id)  
    {
```

```
        this.id = id;  
    }
```

```
    public double getSalary()
```

```
    {  
        return salary;  
    }
```

```
    public String getName()
```

```
    {  
        return name;  
    }
```

```
    public void setName(String name)  
    {
```

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

```
    public void setSalary(double salary)  
    {
```

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

```

package jsp.office
public class Mainclass1
{
    public static void main (String [] args)
    {
        Employee e1 = new Employee (32658, "Ramesh", 250.00);
        SOP (e1.getId());
        SOP (e1.getName());
        SOP (e1.getSalary());
        SOP (" changing object properties");
        e1.setId(26485)
        e1.setName ("Rahul");
        e1.setSalary (350.00);
        SOP (e1.getId());
        SOP (e1.getName ());
        SOP (e1.getSalary ());
    }
}

```

Best example
for Encapsulation
is
JAVA BEANS.

Eclipse
SOURCE
↳ Select
Set & gettor
Automatically
while code

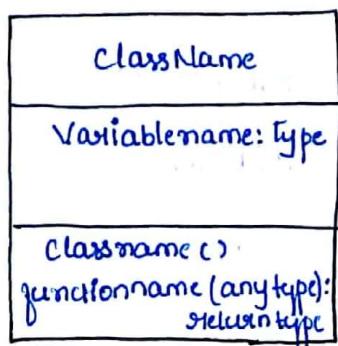
→ Defining a class as a public, private member variables, public constructor

public getter & Setter methods is known as JAVA BEAN CLASS.

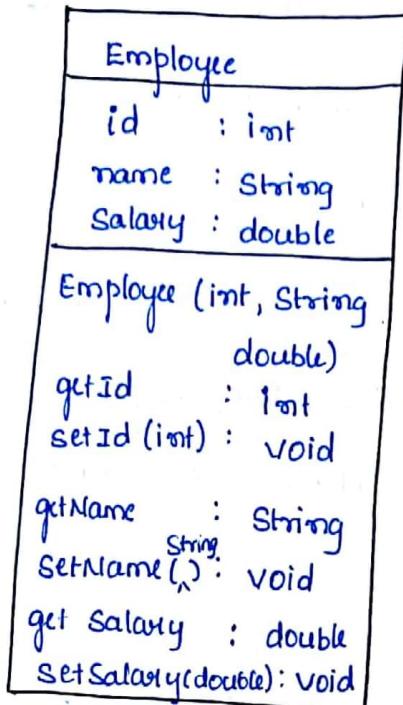
→ The getters methods are used to get the private variables.

→ The setters methods are used to set the private variables values.

- JAVA Bean class is a very good example of Encapsulation.
The class protects the private member variables, the member function provides access to the protected variables.
- In a project JAVA bean class is widely used in implementing below layers
 - ⇒ Data Access object (DAO) Layer
 - ⇒ Data Transfer object (DTO) Layer
- Class Diagram is a pictorial representation of java class and its members.
- It is a part of UML diagram which is used to design the software.
- The class diagram also represents the relationship between the classes.
- When developing a application, first we create class diagram & based on class diagram we write code
- The class diagram consists of 3 Section.
 - 1) represents name of the class
 - 2) represents the list of member variables
 - 3) represents the constructor & Member function of the class



Example:



Marker is a type of Pen

Sketchpen is a type of Pen

Ballpen is a type of Pen

Car is a type of vehicle

BMW is a type of car.

Is-A Relationship

↳ Inheritance (use extends)

or Realization (use implements)

Ex: Class MarkerPen extends Pen

{

}

class MarkerPen implements Pen

{

}

This should be written in main class

06/Sept/2017.

** INHERITANCE (IS-A RELATIONSHIP)

```
class Demo1 {
    int x = 25;
    void test () {
        SOP (" ");
    }
}
```

Class Demo2 extends Demo1

{

int y = 5.8;

void disp()

{

SOP (" ");

Derived class
OR
Subclass
(child class)

Base class
OR
Super class
(parent class)

Demo2 d1 = new Demo2();

SOP (d1.x);

SOP (d1.y);

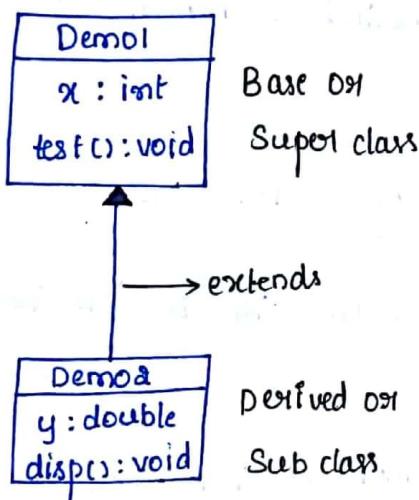
d1.test();

d1.disp();

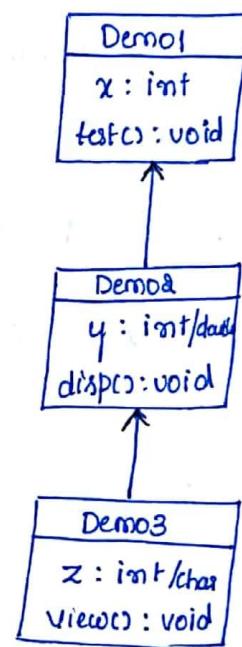
** TYPES OF INHERITANCE



* SINGLE INHERITANCE :

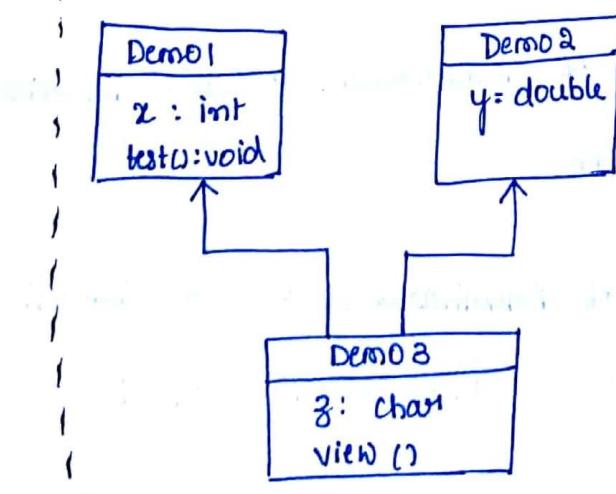


* MULTILEVEL INHERITANCE

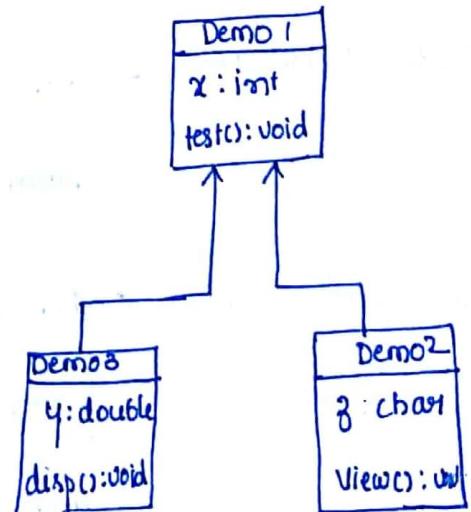


* MULTIPLE INHERITANCE

NOT SUPPORTED.



* HIERARCHIAL INHERITANCE



- Class inheriting from another class is known as Inheritance
- A class can inherit from another class by using extends keyword.
- The class ^{from} where members are inherited is known as BASE class OR SUPER class
- The class to which the members are inherited is known as DERIVED CLASS OR SUB CLASS.
- The Subclass will always have properties of Super class.
- Only the non-static members of Super class will be inherited to the Sub class.
- The static members of a superclass cannot be inherited to the sub classes.
- If the Non-static members of the Super class is having PRIVATE access it cannot be inherited to Sub class bcoz of access restriction.

* Types of Inheritance: There are 4 types of Inheritance.

1. Single Inheritance → In this type of Inheritance, a class inherits from a Superclass
2. Multi-Level Inher → In this type of Inheritance, a class inherits from a Superclass which inherits properties from another Superclass

3. Multiple Inheritance : In this type of inheritance, a class inherits the members

JAVA does not support Multiple Inheritance.

4. Hierarchical Inheritance : In this type of inheritance, more than one Subclass, inherits properties from one Superclass

→ We can combine any of this type of Inheritance, which is known as Hybrid Inheritance.

→ We can always say, a Subclass is a type of Superclass.

→ The CONSTRUCTOR of a Superclass will not be inherited to Subclass but it is used in Inheritance.

→ A class can be declared by using FINAL KEYWORD. Such classes we call it has a FINAL CLASS.

* If class is a final, we cannot inherit define Subclass to it.

* We can create the object of final class & we can also access the members of a final class.

```
-----  
package pack1;
```

```
public class Sample1
```

```
{
```

```
    int x = 24;
```

```
}
```

If we define
class as a final
does not mean
its members
also final.

```

class Sample extends Sample
{
    int x = 45;
    void display()
    {
        SOP ("x value in superclass" + super.x);
        SOP ("x value in subclass" + this.x);
    }
}

```

To access
 Super of super class
 Super-Super.
 x is not
 possible.

```

public class Mainclass {
    PSVM (String [] args) {
        Sample s1 = new Sample ();
        s1.display();
    }
}

O/P: x=24.
      x=45.

```

→ JAVA provides a keyword by name super which is used to access the superclass property in the subclass.

* The keyword must be used only inside the non static method body or constructor body.

<pre> package pack1; public class Demo3 { public int p=23; protected int q=34; } </pre>	<p>Different packages</p> <pre> package pack2; import pack1.Demo3; </pre>	<pre> public class Run1 extends Demo3 { void fun1() { SOP (p); SOP (q); } } </pre>	<pre> package pack3; public class Mainclass { PSVM () { Run1 r1 = new Run1(); r1.fun1(); } } </pre>
-------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------

07-Sep-2017

The benefits of Inheritance are

1) Code Reusability

* Modifications or enhancements.

* Software extensibility.

Note: Every class created in java language must have superclass either defined by compiler or defined by user.

If user does not define any superclass, then the compiler provides default superclass by name Object.

Object class is a Root class in java language and it has around 9 non-static methods.

Every instance created in java language will have Object class properties.

Class Demo extends Object
{
 members
}

Root of java language
9 Methods (Non-static)

Provided by compiler

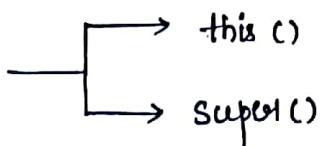
class Sample1 extends Object

{
 =

}
class Sample2 extends Sample1
{
 =

provided by compiler

** CONSTRUCTOR CALLING STATEMENTS



- A Constructor of a class can make a call to the another constructor by using Constructor calling Statements.
- There are two types of Constructor calling Statements If $this()$ or $super()$.
- When a constructor wants to make a call to another constructor of same class, $this()$ statement is used.
- Using $this()$ calling statement, a constructor can call either no arg or parameterized constructor.
- If a constructor wants to call a Super class constructor then we go for $super()$ calling statement.

* Using $super()$ calling statement, the constructor of a class can call either zero arg or parameterized constructor.

→ To use Constructor calling statement we have to satisfy the foll. conditions

- * The Constructor calling statement must be used in the constructor body only.
- * The constructor calling statement must be always first statement of the constructor.
- * Multiple constructor calling statements are not allowed
- * Either $this()$ or $super()$ is allowed but not both together.

```

package pack3;

public class Demo1
{
    int x;
    double y;

    public Demo1 (int x)
    {
        this (12.45);
        SOP ("running Demo1(int)");
        this.x = x;
    }

    public Demo1 (double y)
    {
        SOP ("running Demo1(double)");
        this.y = y;
    }

    void printvalues ()
    {
        SOP ("x value " + x);
        SOP ("y value " + y);
    }
}

----- // Initializing and constructor value
----- in 1st constructor.

package pack3;

public class Mainclass
{
    psvm (string [] args)
    {
        SOP ("Pgm Started");
        Demo1 d1 = new Demo1(25);
        d1.printvalues();
    }
}

```

```

public Demo1 (int x, double y)
{
    this(y)
    SOP ("running Demo1 (int) ");
    this.x = x;
}

public Demo1 (double y)
{
    SOP ("running Demo1 (double) ");
    this.y = y;
}

void printvalues ()
{
    SOP (x);
    SOP (y);
}

```

----- // IMPLICIT CONSTRUCTOR CALLING IN INHERITANCE

```

package pack3

public class Sample1
{
    int z;

    Sample1()
    {
        SOP ("running Sample1()");    x=12;
    }
}

```

```

package pack3;

public class Mainclass1
{
    public static void main (String [] args)
    {
        Demo1 d1 = new Demo1(25,9.5);
        d1.printvalues();

        Demo1 d2 = new Demo1(18,6.7);
        d2.printvalues();
    }
}

```

```
class Sampled extends Sample
```

```
{  
    double y;  
    Sampled (double y)  
    {  
        SOP ("running Sampled (double) constructor");  
        this.y = y;  
    }  
}
```

O/P

```
running Sample () constructor  
running Sampled () constructor  
x value 12  
y value 5.6
```

If a Super class has parameterized constructor, then we need to specify explicitly super()

```
package pack3;
```

```
public class MainClass3
```

```
{
```

```
psvm (String [] args)
```

```
{
```

```
Sample s = new Sample (5.6);
```

```
SOP (s.x);
```

```
SOP (s.y);
```

```
}
```

```
}
```

** EXPLICIT CONSTRUCTOR CALLING IN INHERITANCE

```
package pack3;
```

```
public class Sample1
```

```
{
```

```
int x;
```

```
Sample1 (int x)
```

```

    {
        SOP ("Running Sample1 (int) constructor");
        this.x = x;
    }
}

```

```

class Sample2 extends Sample1
{
    double y;
    Sample2 (double y, int x)
    {
        Super(x);
        SOP ("Running Sample2()");
        this.y = y;
    }
}

```

```

package pack3;

public class Mainclass
{
    psvm (string []args)
    {
        Sample2 s1 = new Sample2
        (5.6, 84);
        SOP (s1.x);
        SOP (s1.y);
    }
}

```

O/P:

Sample1(int)
Sample2(double)

84

5.6

→ Whenever a class inherits from a Superclass, the Subclass constructor must make a call to Super Class Constructor either implicitly or explicitly.

- * If the Super class is having zero or no arg constructor, then the Subclass constructor makes a implicit call to the Superclass.
- * If the Super class is having parameterized constructor, then the subclass has to make a call explicitly.

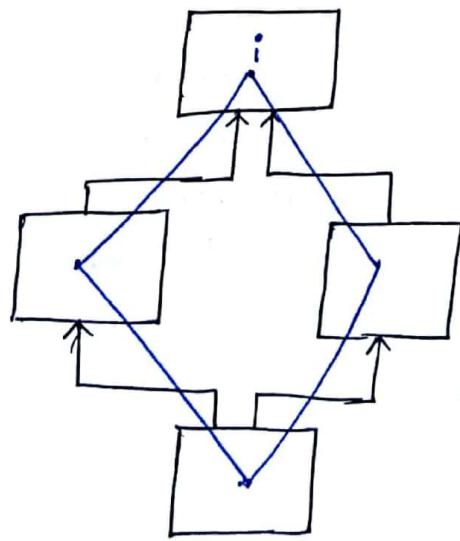
** CONSTRUCTOR CHAINING

Constructor chaining is a phenomenon of running more than one constructor to initialize the states of one object.

- Constructor chaining can happen within one class or it can happen from Subclass to Superclass.
- Whenever we write inheritance program constructor chaining must happen, otherwise inheritance will fail.
- Constructor chaining can be done by this() & super() statements.

08/sep/2017

- JAVA does not support Multiple Inheritance bcoz
 - * Sub class cannot make a call to more than one Super class since multiple calling statements are not allowed.
 - * It leads to an ambiguity known as "DIAMOND PROBLEM".



class S extends G, R
{
 S() {
 Super();
 Super();
 }
}
This is
Not
possible

* * METHOD OVERLOADING:

- Method to perform an operation.
- The same operation as to be performed with different values.
- To achieve compile time polymorphism. OR
- To provide multiple options to perform same operation.

Ex: Online shopping : Delivery after payment

Credit card

Debit card

Ex: Class Demo1

```
{
```

```
void test()
```

```
{
```

```
==
```

```
}
```

```
void test (int arg)
```

```
{
```

```
==
```

```
}
```

Two Rules:

a) Method name must same.

b) Argument must differ

a) Argument type

b) Argument length

Overloading
can be done
for both Static
& Nonstatic
members.

Example for Method Overloading:

```
package pack1;
```

```
public class Demo1
```

```
{
```

// method overloading

```
void test (int arg1)
```

```
{
```

```
    SOP ("running test int argument");
```

```
    SOP ("arg1 value: " + arg1);
```

```
}
```

```
void test (double arg1)
```

```
{
```

```
    SOP ("running double argument");
```

```
    SOP ("arg1 value: " + arg1);
```

```
}
```

```
void test (int arg1, double arg2)
```

```
{
```

```
    SOP ("running (int, double) arg method");
```

```
    SOP ("arg1 value: " + arg1); SOP ("arg2 value: " + arg2);
```

```
} }
```

```
public Mainclass.
```

```
{
```

```
    psvm (String [] args)
```

```
{
```

```
    Demo d1 = new Demo();
```

```
    d1.test(10);
```

```
    d1.test(20, 2.5);
```

```
    d1.test(-1.5);
```

```
} }
```

* Method Overloading can also happens in Subclass.

```
package pack1;

public class Demo2
{
    void disp()
    {
        SOP ("running disp() method");
    }
}

class Demo3 extends Demo2
{
    // method overloading
    void disp (int arg)
    {
        SOP ("running disp(int) method");
        SOP (arg);
    }
}

public class Mainclass2
{
    psvm (String [] args)
    {
        SOP ("Pgm started")
        Demo3 d1 = new Demo3();
        d1.disp();
        d1.disp(25);
    }
}
```

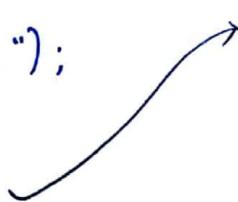
O/P:
running disp() method
running disp (int) method
value is 25.

- In a class defining more than one method with same name and different argument list is known as Method Overloading.
- The argument must differ, in the argument type or in argument length.
- In a class, we can overload the static methods as well as non-static methods.
- The overloaded methods are identified on the basis of method argument.
- The Subclass can overload the methods of Superclass.
- By using method overloading we can achieve Compile-time polymorphism.
- While developing an application, if the same operation or functional has to be performed with different values then we go for Method Overloading. If we want to provide multiple OPTIONS for same function -lity we use Method Overloading.
- Main Method can be Overloaded.

```

public class Mainclass3
{
    public static void main(int args)          O/P. . .
    {
        SOP ("running main int method");
    }
    psum (String [] args)
}
} } { SOP ("running main (String[]) method");

```



11-Sep-2017

** METHOD OVERRIDING.

→ Same operation, but different implementation.
(different way of doing it)

Ex: Swap two numbers

Ex:

```
package pack1;
```

1) Using temp variables

```
public class Demol
```

2) + -

```
{
```

3) / *

```
void test()
```

4) Bitwise

```
{
```

```
SOP ("test () defined in Demol class");
```

```
}
```

```
class Demo2 extends Demol
```

```
{
```

```
void test()
```

```
{
```

```
SOP ("test () overrided in Demo2 class");
```

```
}
```

```
}
```

```
public class Mainclass
```

```
{
```

```
PSVM (String [] args)
```

```
{
```

```
Demol d1 = new Demol();  
d1.test();
```

```
}
```

- Inheriting a method from Superclass and changing its implementation in Subclass is known as METHOD OVERRIDING.
- While overriding, the subclass should retain the same method signature of Superclass.
- To perform Method Overriding IS-A Relationship is compulsory.
- While overriding if we change the method argument, then the method is considered as 'Method Overloading'.
- The following methods of a Superclass cannot be overridden.
 - * static methods → bcoz static methods cannot be inherited to sub class.
 - * final instant method → bcoz final keyword does not allow to change the implementation in Subclass. The final instant method can be inherited to the Subclass.
 - * private instant method → bcoz a private method has a restriction upto class level where it is declared.
- The Method Overriding is used to achieve Run-time Polymorphism.
- While developing an appln. if the old feature or old functionality has to be implemented in a new way then we go for Method Overriding.
- When the same functionality has to be implemented in different ways, then we go for Method Overriding.

** STATIC HIDING:

```
package pack1;

public class Demol
{
    static void testc()
    {
        SOP("testc defined in Demol class");
    }
}

class Demoa extends Demol
{
    static void testc()
    {
        SOP("testc overridden in Demoa class");
    }
}
```

→ The Subclass static method can hide Superclass static method
This is known as STATIC HIDING.

→ While Overriding a method in Sub-class, a Sub-class has a permission to increase the access level but does not have permission to decrease the access level.

* → Constructors can be overloaded but not overriding, bcz it cannot be inherited.

\rightarrow We cannot override MAIN method bcoz it has static &

static method cannot be overriding but can be static hiding.

TYPE CASTING:

→ Converting one type to another type.

1) Primitive Type Casting OR Data Type CASTING.

→ datatype convert to another datatype.

Ex: int converted to double

at widening

float converted to int

b) Narrowing

NON-PRIMITIVE TYPE CASTING OR CLASS CASTING.

→ Class type is converted to another class type

Ex: Demo1 is converted to ~~conversion~~ Demo2.

as Upcasting.

by Downcasting.

* PRIMITIVE TYPE CASTING:

Ex:1) int x = 25 ; → Type Matching

`x` is a variable
of int type

value, of int
type

\Rightarrow double $y = 8.9 \rightarrow$ Type Matching

y is a variable of
discrete type

Value of double type.

Ex 3: $\text{int } x = (\text{int}) 59.99;$ \rightarrow Type Mismatching .

x is a variable
of int type . double type is casted to int .

O/P: $\text{SOP}(x) = \underline{\underline{59}}.$

Ex 4: $\text{double } y = (\text{double}) 125;$ \rightarrow Type Mismatching .

y is a variable of
double type . int type casted to double type .

$\text{SOP}(y) = \underline{\underline{125.0}}$

Ex : package datatypecasting;

public class MainClass

{

psvm (String [] args)

{

SOP ("Program started");

$\text{int } x = (\text{int}) 59.99;$ // double type is casted to int type .

SOP ("x value is :" + x);

$\text{double } y = (\text{double}) 125;$ // int type is casted to double type

SOP ("y value is :" + y);

```

int p = 25 ;
double q = 56.12 ;

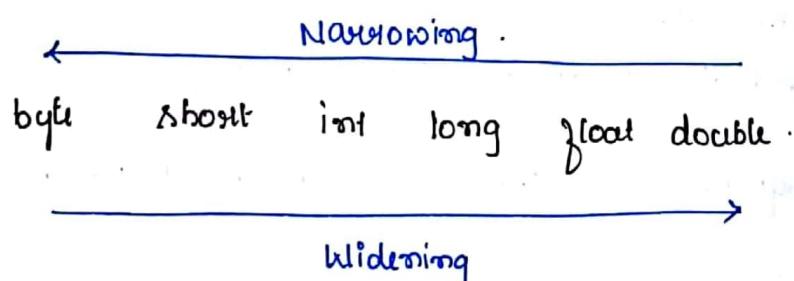
int a = (int) q ; // double type is casted to int type .
double b = (double) p ; // int type is casted to double type

SOP (" a value is :" + a) ;
SOP (" b value is :" + b) ;
SOP (" Program ended") ;
}
}

```

`int x = (int) 59.99 ; // Explicit Narrowing.`

`double y = 125 ; // Implicit Widening or Auto Widening .`



- Converting one type of inform to another type is known as Type Cast
- In JAVA we can perform 2 types of casting. 1> Primitive &
2> Non-primitive
- Type casting .

- Casting one primitive to another primitive type is known as Primitive type casting.
- It is also known as Data type Casting.
- There are two types of Primitive type casting. i) Widening & ii) Narrowing.
- Casting a lower data type to any of the higher data type is known as WIDENING.
- * The Widening can be done either implicitly or explicitly.
- * If Compiler performs the widening it is known as Implicit widening or AUTO WIDENING.
- * If programmer does the widening, then it is known as explicit widening.
- Casting an higher data type to any of the lower data type is known as NARROWING.
- * The Narrowing operation must be done explicitly.
- * Whenever we perform narrowing there will be a loss of data.

```

Ex: package datatypecasting;
public class Calculator
{
    void square (int num)
    {
        SOP ("calculating square " + num);
        int res = num * num;
        SOP ("Result is " + res);
    }
}
  
```

```

package datatypecasting;
public class MainStrg
{
    Calculator C = new Calculator();
    C.square ((int) 5.5);
    double x = 6.5;
    C.square ((int)x);
}
  
```

- While invoking the function, we can perform typecasting for the arguments.
- If the class is having overloaded method, then the first preference will be given for "Type Matching Arguments".

Example for character Type Casting.

```

package datatypecasting;
public class Mainclass {
    public static void main (String [] args)
    {
        char c1 = 'a' ;           int n1 = (int) c1; // character type to int type
        char c2 = 'A' ;           int n2 = (int) c2;   --- we get ASCII values.
        char c3 = 'g' ;           int n3 = (int) c3 ;
        char c4 = 'Z' ;           int n4 = (int) c4 ;

        SOP (c1 + " " + n1);   SOP (c2 + " " + n2); SOP (c3 + " " + n3);
        SOP (c4 + " " + n4);

    }
}
    
```

$a \rightarrow 97 \quad A \rightarrow 65 \quad \sim 32$

$g \rightarrow 122 \quad Z \rightarrow 90 \quad \sim 32$

Alphabet in lowercase - 32 \Rightarrow alphabet in uppercase.

Alphabet in uppercase + 32 \Rightarrow alphabet in lowercase.

** Convert lowercase To uppercase

```
public class Mainclass {
```

```

PSVM (string [] args)
{
    char [] src = {'j', 'a', 'v', 'a'};
    char [] dest = new char [src.length];
    for (int i = 0; i < src.length; i++)
    {
        SOP (src[i]);
    }
}

for (int i = 0; i < src.length; i++)
{
    /*
        int x = (int) src[i]; // ascii value of char
        int y = x - 32; // getting ascii value of upper case char
        dest[i] = (char)y; // copying into dest array .
    */
    dest[i] = (char)(src[i] - 32);
}

SOPm ();
for (int i = 0; i < dest.length; i++)
{
    SOP (dest[i]);
}
}

```

** CLASS TYPECASTING

13-Sep-2017.

```
class Demo1  
{  
    =  
    =  
}
```

```
class Demo2 extends Demo1  
{  
    =  
    =  
}
```

RULES: * Classes must have Is-A Relationship.

* The object of a class must have the properties of another class to which it must be casted.

Demo1 d1 = new Demo1(); → Type Matching.
d1 is the reference → Object of Demo1 type.
variable of Demo1 type.

Demo2 d2 = new Demo2(); → Type Matching.
d2 is the reference → Object of Demo2 type.
variable of Demo2 type.

Demo1 d1 = (Demo1) new Demo2(); → Type Mismatching.
Demo1 type ↓ Demo2 type.
Demo2 type is casted to Demo1 type.

Demo2 d2 = (Demo2) new Demo1(); → Type Mismatching.
Demo2 type ↓ Demo1 type.
Demo1 type is casted to Demo2 type.

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

```
Demo1 d1 = (Demo1) new Demo1(); // Upcasting → Subclass  
type is casted to Super class type.
```

```
Demo d2 = (Demo2) new Demo1(); // Down casting: Superclass  
type is casted to Sub class type.
```

```
}
```

→ Casting: One class type to another class type is known as CLASS-TYPE CASTING.

→ To perform class-type Casting the following rules should be met.

- * The classes must have Is-A relationship.

- * The class or the object must have the properties of the class to which it must be casted.

→ If the first rule is not satisfied, the COMPILER throws error.

→ If the second rule is not satisfied, the JVM throws classcast exception.

Exception:

→ There are two types of Casting

Up casting.

Down Casting.

→ Casting Subclass type to Superclass type is known as UP CASTING.

- * Up casting can be done either implicitly or explicitly.

- * If compiler performs up casting, it is known as Implicit up-casting.

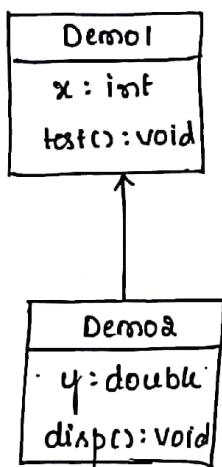
- * If code performs up casting, it is known as Explicit up-casting.

→ Casting Superclass type to Subclass type is known as DOWN CASTING.

* Down casting has to be done explicitly.

* The down casting should be done only on the object which is already upcasted.

* UPCASTING



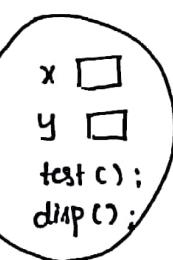
Demo d1 = (Demo) new Demo2();

SOP(d1.x);

d1.test();

When it is upcasted,

The Subclass completely behaves like Superclass (in this ex. Demo1 is Superclass).



Demo2 object

Example :-

```
public class Demo1
{
    int x = 12
    void test()
    {
        SOP("running test() method");
    }
}
```

```
public class Demo2 extends Demo1
{
    double y = 5.7;
    void disp()
    {
        SOP("running disp() method");
    }
}
```

PSVM (String [] args)

{

 Demo1 d1 = (Demo1) new Demo2(); // upcasting.

 SOP (d1.x); ✓

 d1. test(); ✓

 SOP (d1.y); X

 d1. disp(); X

}

 } ↓
 } ↓
 } ↓
 } This completely behaves like Demo3 (Superclass)

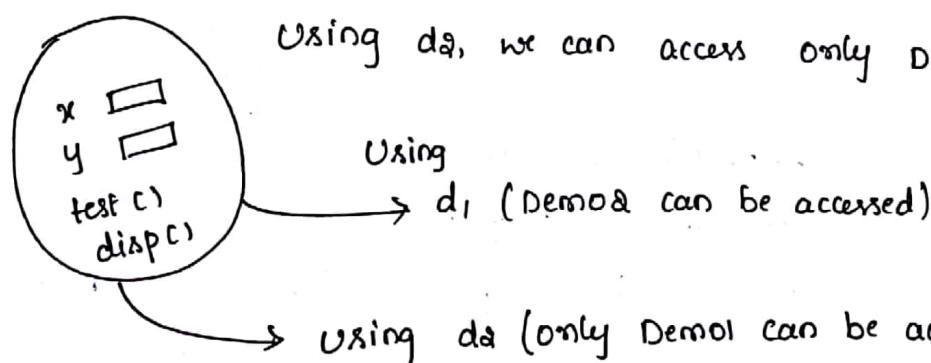
Example 2:

 Demo2 d1 = new Demo2();

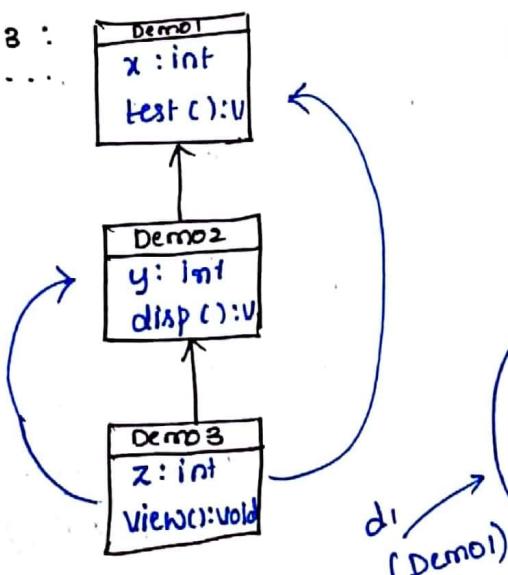
 Demo1 d2 = (Demo1) d1;

Using d1, we can access Demo1 & Demo2 properties

Using d2, we can access only Demo1 properties.



Example 3:

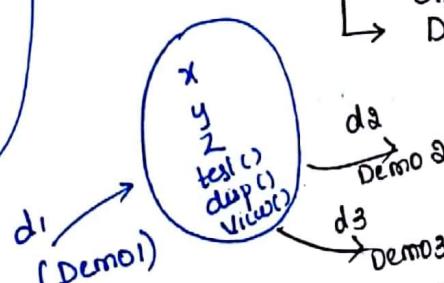


 Demo1 d1 = new Demo3();

 ↳ Access only Demo1 properties.

 Demo2 d2 = new Demo3();

 ↳ Using d2
 Demo2, Demo1 properties
 are accessed.



{
 `psvm (String [] args) {

Implicit upcasting: Using d1 we can access Demo1 // Demo1 d1 = new Demo3();

Down Casting : Demo3 & Demo1 // Demo3 d2 = (Demo3)d1;

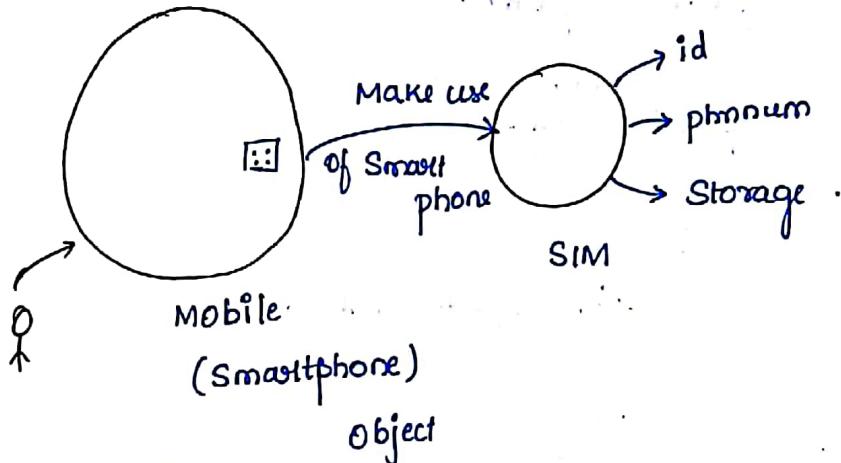
Down casting : Demo3 properties // Demo3 d3 = (Demo3)d1;
 ↓
 Demo1 & Demo2 & Demo3 :

}

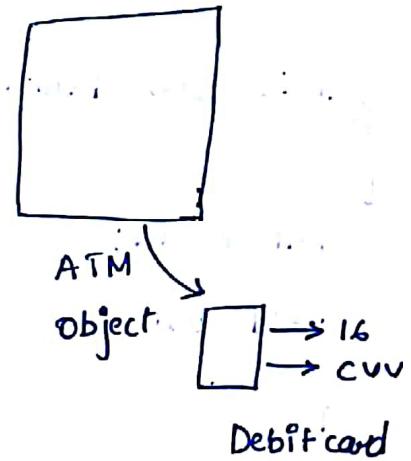
14/09/2017

** OBJECT COMMUNICATING WITH ANOTHER OBJECT :

Ex1: Mobile & SIM



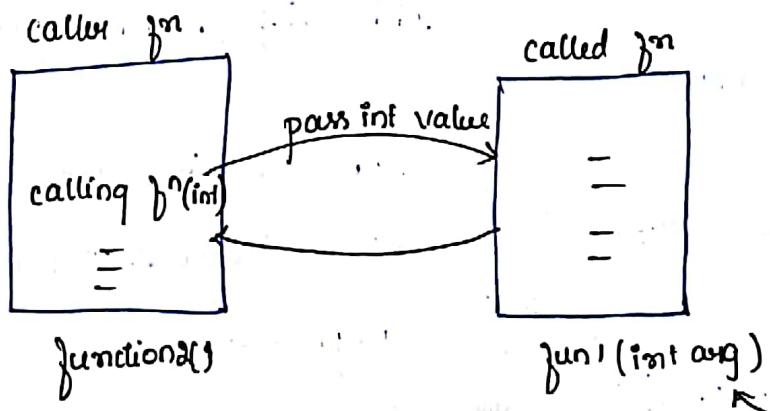
Ex2: ATM AND CARD.



* FUNCTIONS → WITH CLASS TYPE ARGUMENTS (*****)

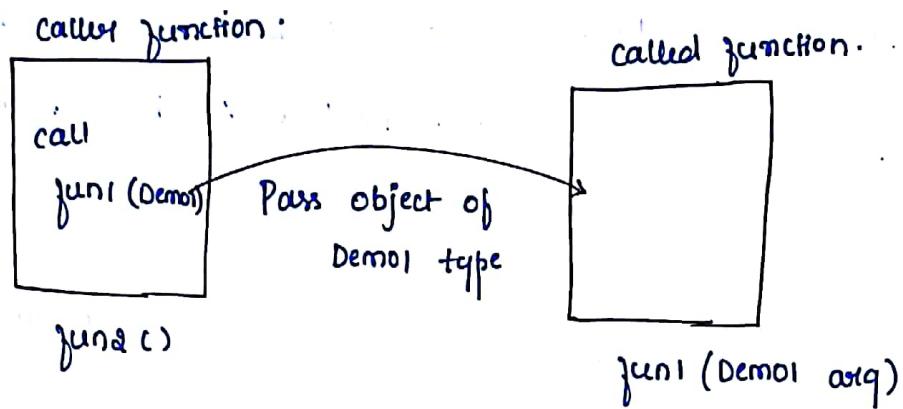
Ex: Primitive type calling.

- perform operation
- Reusability
- Modularity



Scanned by CamScanner

Example 2 : Non-primitive Type (class Type) :



Ex:

```
package pack1;
```

```
public class Demo1  
{
```

```
    int x=12;
```

```
    void test()
```

```
{  
    System.out.println("Running test()");  
}
```

```
SOP ("Running test()");  
}
```

```
package pack1;
```

```
public class Sample1  
{
```

```
    void square (int num)
```

```
{
```

```
    int res = num*num;
```

```
    SOP ("Square is "+res);  
}
```

```
// fun argument type is Demo1 type.
```

```
void disp (Demo1 obj)
```

```
{
```

```
    SOP ("Running disp() method ...");
```

```
    SOP ("x value: " + obj.x);
```

```
    obj.test();  
}
```

```
PSVM (String [] args)
```

```
{
```

```
    Sample1 s1 = new Sample1();
```

```
    s1.square(5); // calling fn, passing int  
    // value
```

```
    s1.disp (new Demo1()); // calling  
    // fn, passing Demo1 class ob
```

```
}  
}
```

OR

psvm (String [] args)

Sample s1 = new Sample();

int a = 5;

s1.square(a);

Demol d1 = new Demol();

s1.disp(d1); // calling gm, passing object reference.

Example 2

public class Sample

{
void square (int num)

{
int res = num * num;

SOP ("Square is " + res);

}

// gm argument type is Demol type

void disp (Demol arg1)

O/P

{
SOP ("Invoking disp () method");

SOP ("X value is : " + arg1.x);

arg1.test();

The value of x is 150:
arg1.x = 150;

}

psvm (String [] args)

{

}

SOP ("x value : " + d1.x); // ??

→ Function argument can be primitive or Non-primitive type.

* If the argument is primitive type, the caller function has to pass the value to the called function.

* If the argument is Non-primitive type, then the caller fn has to pass the object of the class type.

* The called function can make use of object properties to perform operations, it can also update the object properties.

→ The function return type can be primitive type or Non-primitive type.

* If the return type is primitive, the junction has to return primitive value.

* If the return type is non-primitive (class type), the junction has to return the object of class type mentioned in return type.

15/09/2017

Q) Design and write Java program for the following requirement

Employee has following properties:

1) Name

2) Id

3) Salary

The finance team of an organization is responsible for increasing the salary of an employee by 5%.

```
package pack1;

public class Employee
{
    int id;
    double salary;
    String name;

    public Employee (int id, double salary, String name)
    {
        this.id = id;
        this.salary = salary;
        this.name = name;
    }
}
```

```
package pack1 // Finance uses Employee property to inc salary.

public class Finance
{
    void incrementSalary (Employee emp, int incRate)
    {
        System.out.println ("incrementing salary of employee " + emp.name);
        emp.salary = emp.salary + (emp.salary * incRate) / 100;
    }
}
```

```
package pack1;

public class MainClass3 {
    public static void main (String [] args) {
        Employee e1 = new Employee (3215, 25000.00, "Raju");
        Finance f1 = new Finance ();
        SOP ("Employee Salary: " + e1.salary);
        f1.increaseSalary (e1, 15);
        SOP ("Employee Salary: " + e1.salary);
    }
}
```

** GENERALISATION:

```
package pack1;

public class Demo1 {
    int x = 12;

    void test () {
        System.out.println ("running test() method");
    }
}
```

```
package pack1;

public class Demo2 extends Demo1
{
    double y = 12.34;

    void view()
    {
        SOP("running disp() method ...");
    }
}
```

```
package pack1;

public class Demo3 extends Demo1
{
    char x = 'a';

    void disp()
    {
        SOP("running disp() method");
    }
}
```

```
package pack1;

public class Sample1
{
    // Specialization
    void func1 (Demo2 arg)

```

```
package pack1

public class Sample1
{
    // Generalization
    void func1 (Demo1 arg)

```

```

{ }

SOP ("Running func() method ...");
SOP ("x value : " + arg.x);
SOP ("y value: " + arg.y);
arg.test();
arg.view();
}

package pack1;
public class Mainclass
{
    psvm (String [] args)
    {
        SOP ("main method started");
        Demo1 d1 = new Demo1();
        Demo2 d2 = new Demo2();
        Demo3 d3 = new Demo3();
        Sample s1 = new Sample1();
        s1.func1 (d1);
        SOP (".....");
        s1.func1 (d2);
        SOP (".....");
        s1.func1 (d3);
    }
}

```

```

{ }

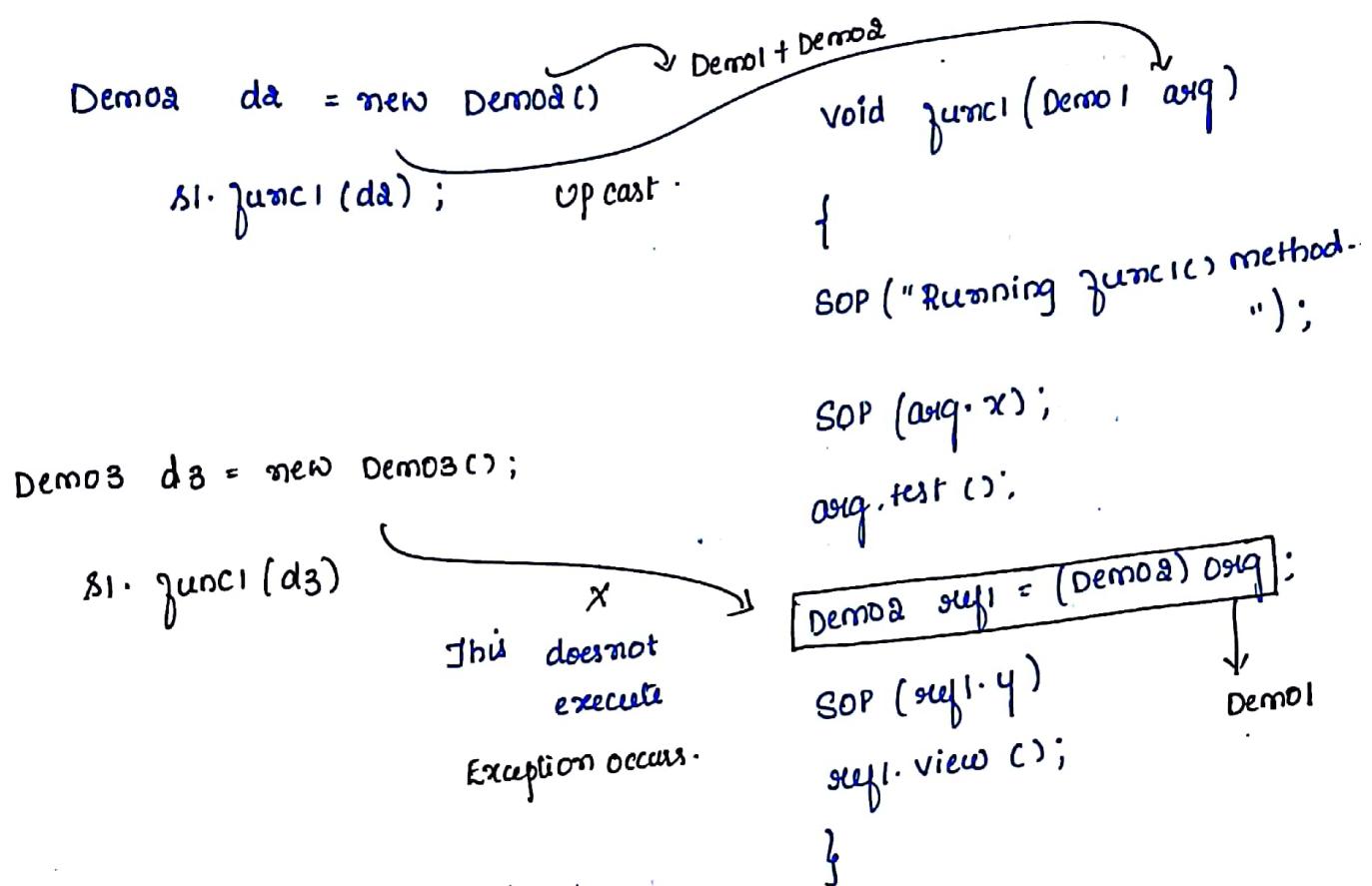
SOP ("Running func() method ...");
SOP ("x value : " + arg.x);
arg.test();

}

package pack1;
public class Mainclass
{
    psvm (String [] args)
    {
        Demo1 d1 = new Demo1();
        Demo2 d2 = new Demo2();
        Demo3 d3 = new Demo3();
        Sample s1 = new Sample1();
        s1.func1 (d1);
        SOP (".....");
        s1.func1 (d2);
        SOP (".....");
        s1.func1 (d3);
    }
}

```

** Accessing the specific properties by performing Downcasting



```

package pack1;

public class Sample1
{
    // Generalization

    void func1(Demo1 arg)
    {
        SOP(" Running func1 method....");

        SOP(" x value: " + arg.x);
        arg.test();

        if (arg instanceof Demo2) {
            Demo2 ref1 = (Demo2) arg;
        }
    }
}
  
```

```

SOP ("y value :" + ref.y);
ref.view();
}

else if (arg instanceof Demo3)
{
    Demo3 ref = (Demo3) arg;
    SOP ("x value :" + ref.x);
    ref disp();
}
}

```

16/09/2017

→ Defining the junctions which runs for different type of objects is known as Generalisation.

→ The junction can be called by passing the object having the properties of the class defined in the junction argument.

→ Any object which has that properties can be given to the junction.

→ Defining the junction which works for only one type of object is known as Specialization.

* In this case the junction execution happens only when we give the object is exact type of the class mentioned in the junction argument.

Q) Design and Develop a java program for the below requirement.

A citizen of a country has two properties i.e Age & Name.

An employee is a citizen having employee ID & Salary. A student is also a citizen having Rollno and Marks. The Govt is providing following services

* Enclosing to Aadhar No → Applicable to all the citizen.

* Accoding Scholarship → Applicable to all the merit students.

* Income Tax → Applicable for all the employees whose monthly salary is 45K and above.

```
package pack1;
```

```
public class citizen
```

```
{
```

```
int age;
```

```
String name;
```

```
}
```

```
public citizen (int age, String name)
```

```
{
```

```
this.age = age;
```

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

```
}
```

```
}
```

Public class employee extends citizen

```
{
```

```
package pack1;

public class Government
{
    void aadharEnrollment (Citizen arg)
    {
        SOP ("Aadhar Enrollment success for " + arg.name);
    }

    void incomeTax (Employee arg)
    {
        SOP ("calculating Tax");
        // income tax - 30%
        double taxAmt = 0.0;
        if (arg.salary >= 45000.00)
        {
            taxAmt = (arg.salary * 30) / 100;
            SOP ("Tax Amount is " + taxAmt);
        }
    }

    void scholarship (Student arg)
    {
        if (arg.marks > 70.00)
        {
            SOP ("Scholarship awarded to " + arg.name);
        }
    }
}
```

```
public class Mainclass
```

```
{
```

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

```
{
```

```
    SOP ("*****");
```

```
    Citizen c1 = new Citizen (25, "Raju");
```

```
    Student s1 = new Student (21, "cat", 125, 75.52);
```

```
    Employee e1 = new Employee (28, "Vimod", 480, 50000.00);
```

```
    Government gov = new Government();
```

```
    gov. AadharEnrollment (c1);
```

```
    gov. AadharEnrollment (s1);
```

```
    gov. AadharEnrollment (e1);
```

```
    gov. Scholarship (s1);
```

```
    gov. IncomeTax (e1);
```

```
    SOP ("*****");
```

```
}
```

```
}
```

POLYMORPHISM.

- An object showing different behaviors at different stages of its life cycle is known as POLYMORPHISM.
- There are two types of Polymorphism



→ In Compile-time Polymorphism is binded to the method definition by the compiler at the time of compilation.

* Since the binding is done during compilation, it is known as Early Binding.

* Since compiler binds the method, it cannot be subinduced, hence it is known as STATIC BINDING.

* Method Overloading is an example to achieve Compile-time polymorphism. Here the binding happens on the basis of arguments.

→ In Runtime Polymorphism, the method declaration is binded to the method defn by JVM during execution.

* Since the binding is happening at execution time, it is known as LATE BINDING.

* The binding done by the JVM can be subinduced. Hence it is known as DYNAMIC BINDING.

* Method Overriding is used to achieve Runtime polymorphism.

* In order to write Run-time polymorphic code we have to satisfy following principles

* Is-A Relationship

* Method Overriding

* class Typecasting

** MEMORY USED BY JVM:

Whenever we execute program in JVM, the JVM uses the following memory area.

* Heap Area

* Class Area

* Stack Area

* Method Area

* HEAP AREA:

The JVM stores all the objects created in the code in Heap Area.

Memory is allocated randomly in Heap Area.

→ Whenever the object is created, the object details (state & behaviour) is loaded in the memory in the hash table structure

HASH TABLE

Identifiers	Value
v.name	value
v.name	value
f.name()	value
f.name()	
:	:

** CLASS AREA

- Class Area is used to store only static members of the class.
- All the static members of the class are loaded into the static pool created in the class Area.
- A static pool is a bunch of memory, which can store only static members.
- The pool name will be same as the class name.

* METHOD AREA

- Method area is used to store only the defn or method body of the class.
- If the method declaration is NON-STATIC, then the declaration part is loaded in the heap memory.
- If the method declaration is STATIC, then the method declaration part is loaded in the class area.

* STACK AREA :

- The stack area is used for execution purpose.
- The function execution happens on the stack.
- Stack follows LIFO Structure.

When a fn or method called for execution, the called fn comes on top of the stack & begins its execution.

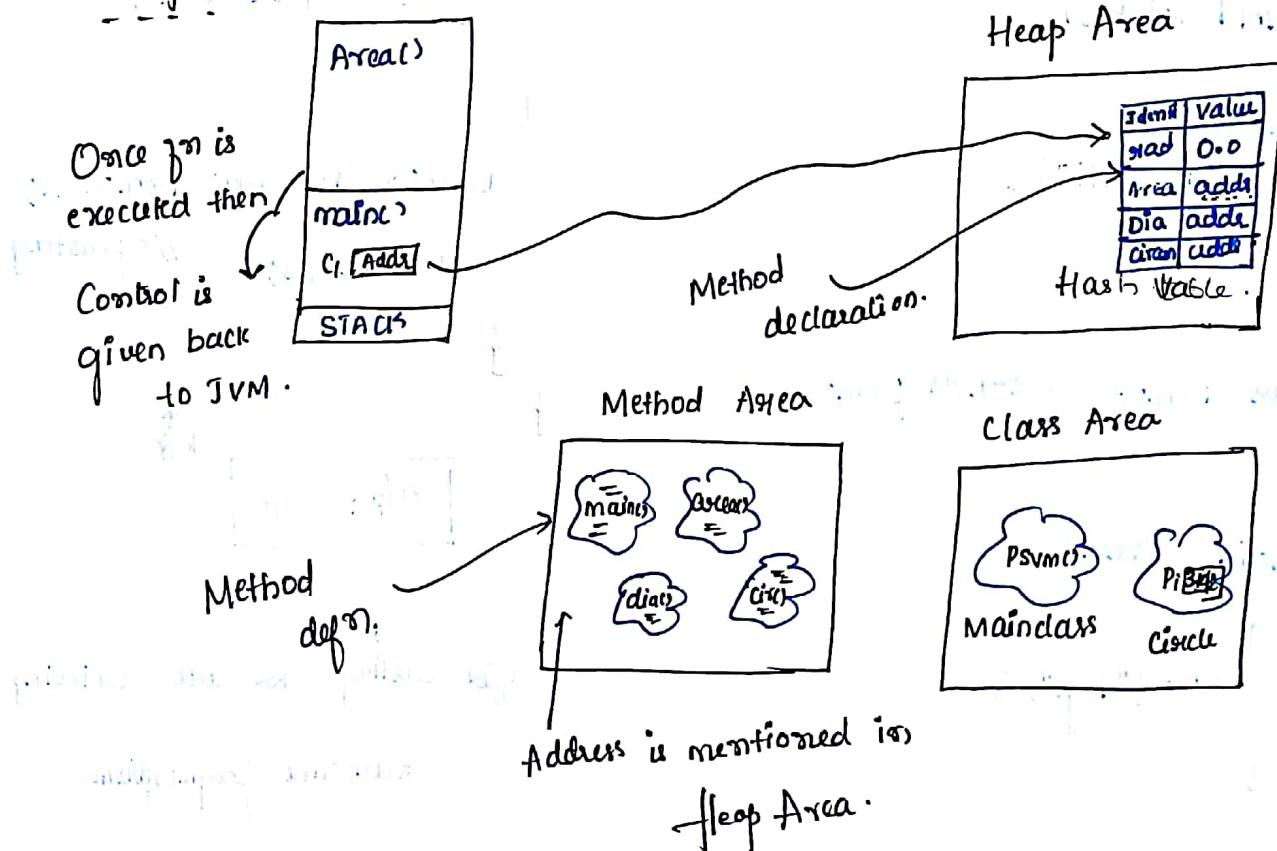
→ If the `fn` has local variables, then it is stored in the stack memory area.

CLASS LOADER of JVM is responsible for loading the class and its members into the memory.

→ The class loader loads always static members first, while loading the class members it ignores the static members of class.
→ The Non-static members are loaded only when the object is created while loading the non-static members, the class loader loads the static members first, if static members are not loaded. If it is already loaded, it will not load again.

18/sep/2017

Example:



Example 2: Run time Polymorphism . (With Arguments) .

→ If matches the address by
the type of arguments

Identifier	Value
test()	address
test(int)	address
test (double)	address

Example 3: Overriding Concept

```
package pack1;
```

```
public class Demo2 {
```

```
{  
    void wish()  
    {  
        System.out.println("Hi");  
    }  
}
```

```
class Demo3 extends Demo2 {
```

```
{  
    void wish()  
    {  
        System.out.println("Bye");  
    }  
}
```

```
package pack1;
```

```
public class Mainclass {
```

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

```
        Demo2 d1 = new Demo3();  
        d1.wish(); // Upcasting  
    }
```

O/p : Bye.

After casting, we are achieving
Run time polymorphism.

** Example for Types of Polymorphism:

19/09/2017

```
package pack1;
```

```
public class Animal
```

```
{
```

```
void main()
```

```
{
```

```
SOP("All Animal makes noise");
```

```
}
```

```
}
```

```
-----
```

```
package pack1
```

```
public class Dog extends Animal
```

```
{
```

```
void noise()
```

```
{
```

```
SOP(" Bow.... Bow");
```

```
}
```

```
}
```

```
-----
```

```
package pack1
```

```
public class AnimalSimulator
```

```
{
```

```
void makeNoise(Animal arg)
```

```
{
```

```
arg.noise();
```

Generalization.

```
-----
```

```
package pack1
```

```
{
```

```
void noise()
```

```
{
```

```
SOP(" Hiss.... Hiss...");
```

```
}
```

```
}
```

If cat, or
dog, or
Snake called
then it becomes
Specialization.

```

package pack1

public class TestAnimalApp

{
    public (String [] args)
    {
        Cat c1 = new Cat();
        Dog d1 = new Dog();

        Snake s1 = new Snake();
        AnimalSimulator a1 = new AnimalSimulator();

        a1.makentaise(s1);           → Generalized argument
                                     (d1) OR (c1);
    }
}

```

→ we can also call Superclass by creating object

** ABSTRACTION :

* ABSTRACT METHOD :

- Methods without definition / body.
- Terminate at declaration time.
- Are declared using Abstract.
- Abstract method must be declared only in abstract class

OR In INTERFACE.

Syntax:

```

abstract returnType MethodName
                    (Arguments)

```

* CONCRETE METHOD:

- Methods having both declaration and definition.
- Concrete methods can be developed in or Non-Abstract class
or Concrete class
or Abstract class.

* Syntax:

```
returntype methodname (Arguments)  
{  
}  
-----
```

- Declaring a class using abstract keyword is known as Abstract class.
- In a abstract class we can define static & non static variables.
- We can define a constructor in a abstract class.
- We can define both static functions & Non-static functions.
- If class is abstract, it is not compulsory to declare Abstract methods.
- In a Abstract class, we can define both Abstract methods & concrete methods.
having only
- Methods declaration is known as ABSTRACT METHOD.
- The Abstract Methods are declared by using abstract keyword.

→ The abstract method must be declared either in Abstract class OR in JAVA Interfaces.

- * If a class is declared as Abstract, we cannot create the object of the class.
- * The static members of Abstract class can be accessed by using class name, we cannot access non-static members, ^{bcoz} we cannot create object of Abstract class

Example : abstract public class Demo1

```
{

    static int x=12;           → Static Member Variables.

    int y;

    Demo1 (int y)

    {

        this.y=y;

    }

    static void test()

    {

        SOP ("running test ()");

    }

    void disp()

    {

        SOP ("running disp ()");

    }

}
```

```
public class Mainclass
```

```
{
```

```
    PSVM (String [] args)
```

```
{
```

```
    SOP ("x value is :" + Demo1.x);
```

```
    Demo1.test();
```

```
}
```

```
}
```

Ex 2: Inheritance in Abstract class.

```
package pack1;
```

```
public abstract class Demo2
```

```
{
```

```
    int x=12;
```

```
    void test()
```

```
{
```

```
    SOP ("Running test()");
```

```
}
```

```
}
```

```
class Sample1 extends Demo2
```

```
{
```

```
    int y=10;
```

```
}
```

```
public class Mainclass
```

```
{
```

```
    PSVM (String [] args)
```

```
{
```

```
    Sample s1 = new Sample1();
```

```
    SOP (s1.x);
```

```
    s1.test();
```

- - - - // we can also use Reference
of abstract class.

```
Demo d1 = new Sample();
```

// upcasting.

```
SOP (d1.x);
```

```
d1.test();
```

→ The abstract keyword cannot be combined with following keyword

* static
* final
* private

Error:
Illegal combination.

20/09/17

Any method that cannot be overridden, can also be declared using abstract.

Example 18 : package pack1;

```
public abstract class Demo2
```

```
{
```

```
    int x = 12;
```

```
    void test()
```

```
{
```

```
        SOP("running test() method ...");
```

```
}
```

```
    abstract void disp();
```

```
}
```

abstract class Sample1 extends Demo2

```
{
```

```
}
```

```
class Sample1 extends Demo2
```

```
{    void disp()
```

```
    {        SOP("disp() running in  
            Sample1");    }
```

Since Subclass is inheriting
the abstract method
the Subclass must be
declared as 'abstract':

If the class want to remain
without abstract then Subclass
has to provide the defn to
abstract method.

```
public class Mainclass  
{  
    public void (String [] args)  
    {  
        Sample s1 = new Sample();  
        SOP (s1.x);  
        s1.test(); s1.disp();  
  
        Demo2 d2 = new Sample(); // up-casting  
        SOP (d2.z);  
        d2.test(); d2.disp();  
    }  
}
```

Example: Defining abstract class Methods in different Subclass.

```
abstract public class Demo3  
{  
    abstract void test();  
    abstract void disp();  
}
```

```
Abstract class Sample2 extends Demo3  
{  
    void test()  
    {  
        SOP ("test() defined in Sample2 class");  
    }  
}
```

```
class Sample3 extends Sample2  
{  
    void disp()  
    {  
        System.out.println("disp() running in Sample3 class");  
    }  
}
```

```
public class MainClass3  
{  
    public static void main (String [] args)  
    {  
        Sample3 s1 = new Sample3();  
        s1.test();  
        s1.disp();  
    }  
}
```

→ If a class builds a Is-A relationship with abstract class,

the Subclass must provide a definition to all the abstract methods of the abstract class. If not the Sub class must be declared as Abstract class.

→ This is known as Contract of Abstract.

```
package pack1;

abstract public class DemoA
{
}

class Sample4 extends DemoA
{
}

// Sample4 is concrete class, because the abstract class
// does not have abstract methods
```

→ A class defining only abstract methods is known as PURE ABSTRACT BODY.

```
** abstract class Demo1
{
    abstract void m1();
    abstract void m2();
}
```

// This is not pure Abstract body
because it is getting inheriting
from Object class.

```
** abstract public class Demo5
{
    int x;
    Demo5(int x)
    {
        this.x = x;
    }
}
```

```

abstract void test();

}

class Sample5 extends Demo5
{
    Sample5 (int x)
    {
        Super(x);
    }

    void test ()
    {
        SOP ("running test method");
    }
}

public class Mainclass
{
    public static void main (String [] args)
    {
        Sample5 s1 = new Sample (85);

        s1.test ();
    }
}

```

construction chaining
concept.

Here we are fulfilling
Inheritance + Abstraction
Rules }

** JAVA INTERFACES

interface InterfaceName → declaration
{ }

Only static variables and it must be final

No Constructors

Only Abstract Methods

Only public access is allowed

} Body of the Interface.

Example: package pack1;

public interface Demo1

{

int x = 123;

Automatically it considers as

Static final int x=123

void test();

}

Keyword.

Class Sample1 implements Demo1

{

public void test()

{

Access specifier must be public
since visibility cannot be decreased.

System.out.println("test implemented in Sample1 class");

}

}

```
psvm ( string [] args )
{
    Sample s1 = new Sample();
    s1.test();
    SOP ("-----");
}

Demo d1 = new Sample();
d1.test();

}
```

→ JAVA Interface is a type defn block used to define only Abstract Methods.

→ In the Interface body we can declare static variables. & these variables must be final. It must be initialized @ declaration time only.

→ We cannot define a constructor inside the Interface.

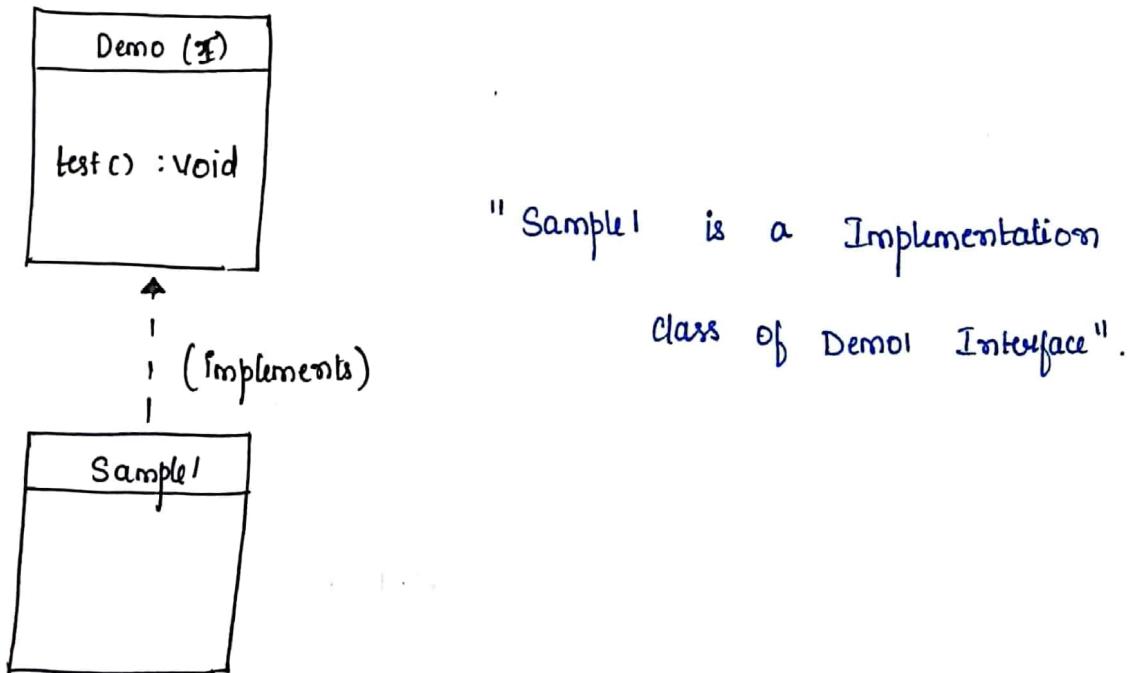
→ Interface allows only abstraction method declaration, It does not allow to define concrete method.

→ Interface allows only public access.

→ We cannot create Object of the interface type, we can declare Reference variable of Interface type.

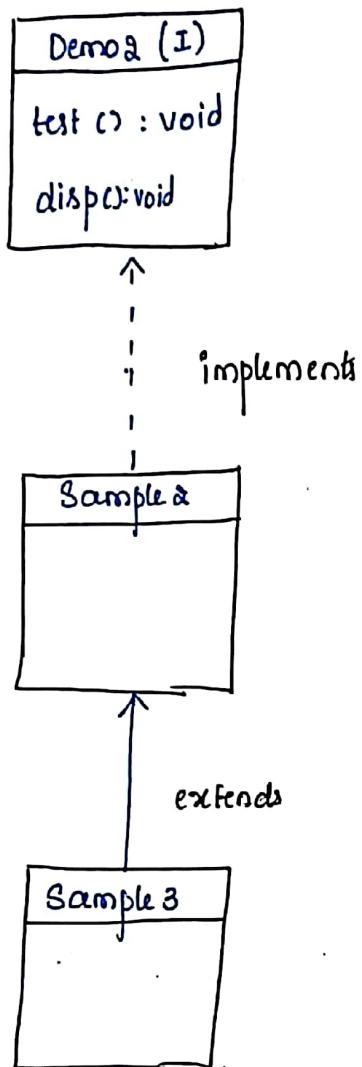
→ An Interface can extend from another Interface but not from class.

- The methods of the interface are implemented in the class.
 - A class can implement Interface method by using implement keyword.
 - When a class implements Interface, a class must provide implementation to all the abstract methods of the Interface, if not the class must be declared as Abstract.
 - A class can implement more than one Interfaces.
 - A class which implements an Interface Method is known as Implementation class.
-



** Implementation Class

all solution



```
package pack1;
```

```
public interface Demo2
{
    void test();
    void disp();
}
```

```
abstract class Sample2 implements Demo2
```

```
{
    public void test()
    {
        System.out.println("test() implemented in Sample2");
    }
}
```

```
class Sample3 extends Sample2
```

```
{
    public void disp()
    {
        System.out.println("disp() implemented in Sample3
class");
    }
}
```

```
}
```

```
| package pack1  
|  
| public class Mainclass  
|  
| {  
|    public void main (String [] args)  
|    {  
|        Sample3 S1 = new Sample3()  
|  
|        S1.disp ();  
|        S1.test ();  
|  
|        Demo3 d1 = new Sample3();  
|        d1.test ();  
|        d1.disp ();  
|    }  
| }  
| }
```

```
| Example: package pack1;  
|  
| public interface Demo3  
|{  
|    void test();  
|}  
|  
| interface Demo4 extends Demo3  
|{  
|    void disp();  
|}
```

```

class Sample4 implements Demo4
{
    public void test()
    {
        System.out.println("test() implemented in Sample4 class");
    }

    public void disp()
    {
        System.out.println("disp implemented in Sample4 class");
    }
}

```

```

package pack1;

public class MainClass3
{
    public static void main(String[] args)
    {
    }
}

```

```

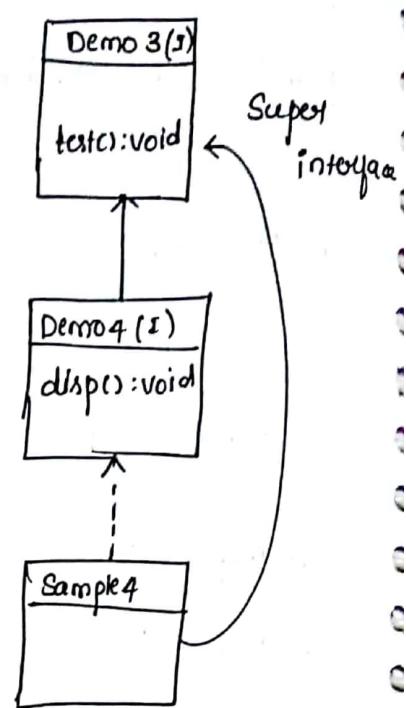
Demo4 d1 = new Sample4();
d1.test();
d1.disp();

```

```

Demo3 d2 = new Sample4();
d2.test();
d2.disp(); // throws error Demo3 does not have property of D4.
}

```



** Multiple Implementation is possible in JAVA (REALIZATION)

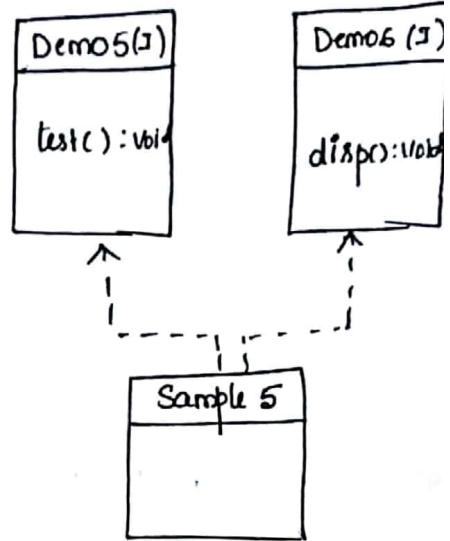
```
package pack1;

public interface Demo5
{
    void test();
}

interface Demo6
{
    void disp();
}
```

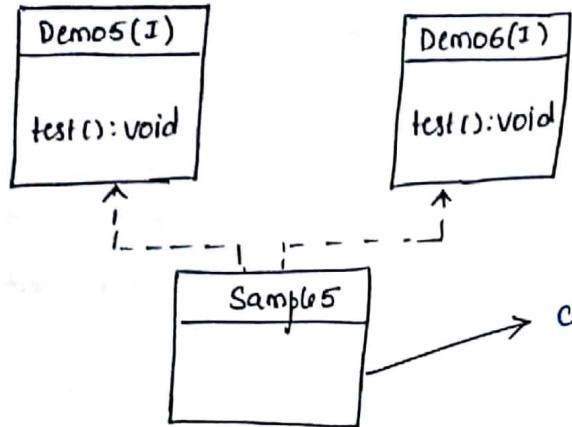
```
class Sample5 implements Demo5, Demo6
{
    public void test()
    {
        SOP ("test() is implemented in Sample5 class");
    }

    public void disp()
    {
        SOP ("disp() is implemented in Sample5 class");
    }
}
```



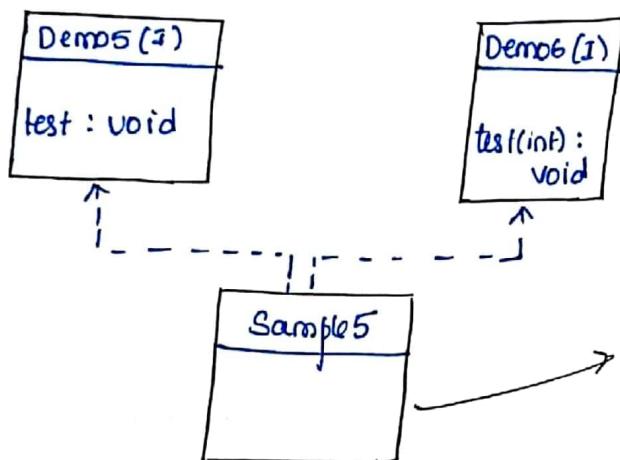
```
public class Mainclass4
{
    PSVM (String [] args)
    {
        Demo5 d1 = new Sample5();
        d1.test();
        SOP (".....");
        Demo6 d2 = new Sample5();
        d2.disp();
    }
}
```

Q7



class must provide
One implementation.

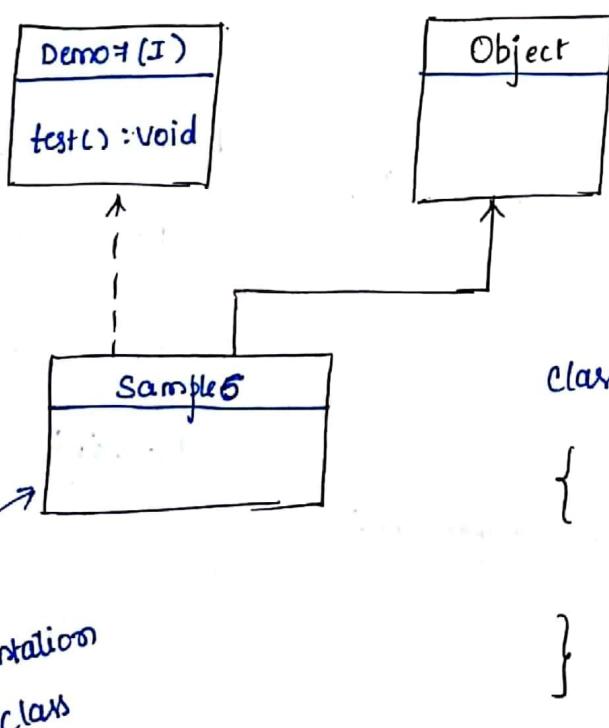
Q8



class must provide
two implementations.

Overloaded test() methods.

Q9

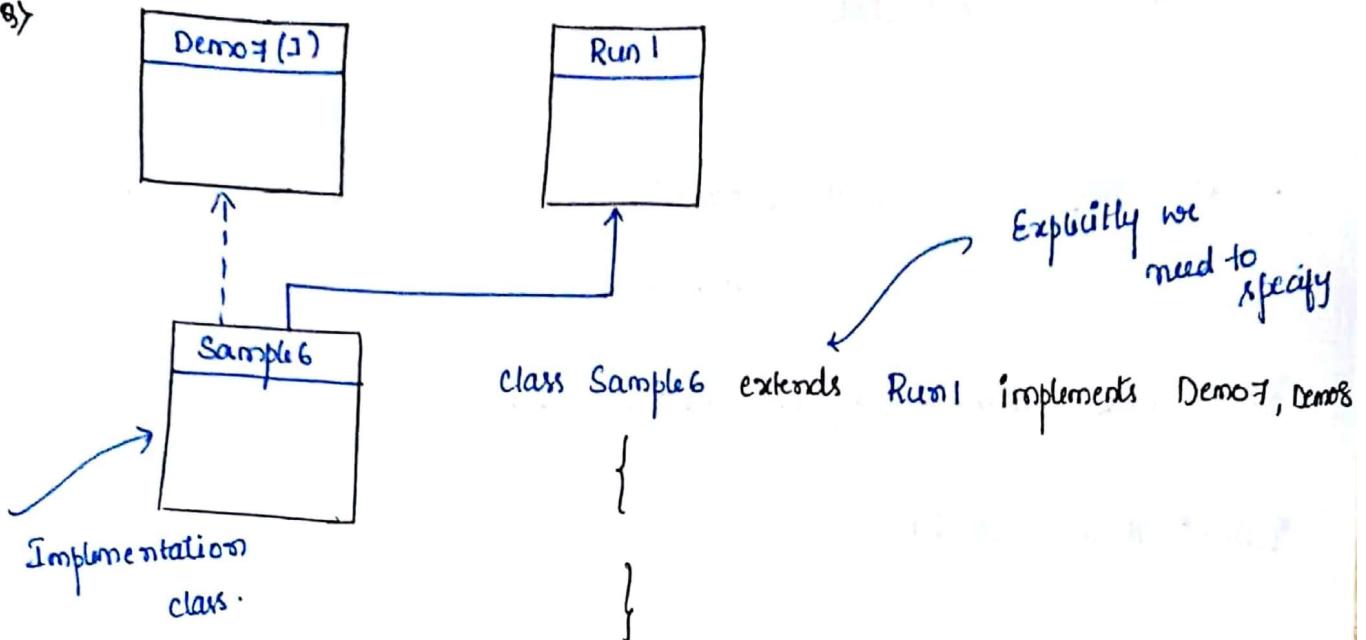


Class Sample5 extends Object implements
Demos(I)

compiler
writes
implicitly

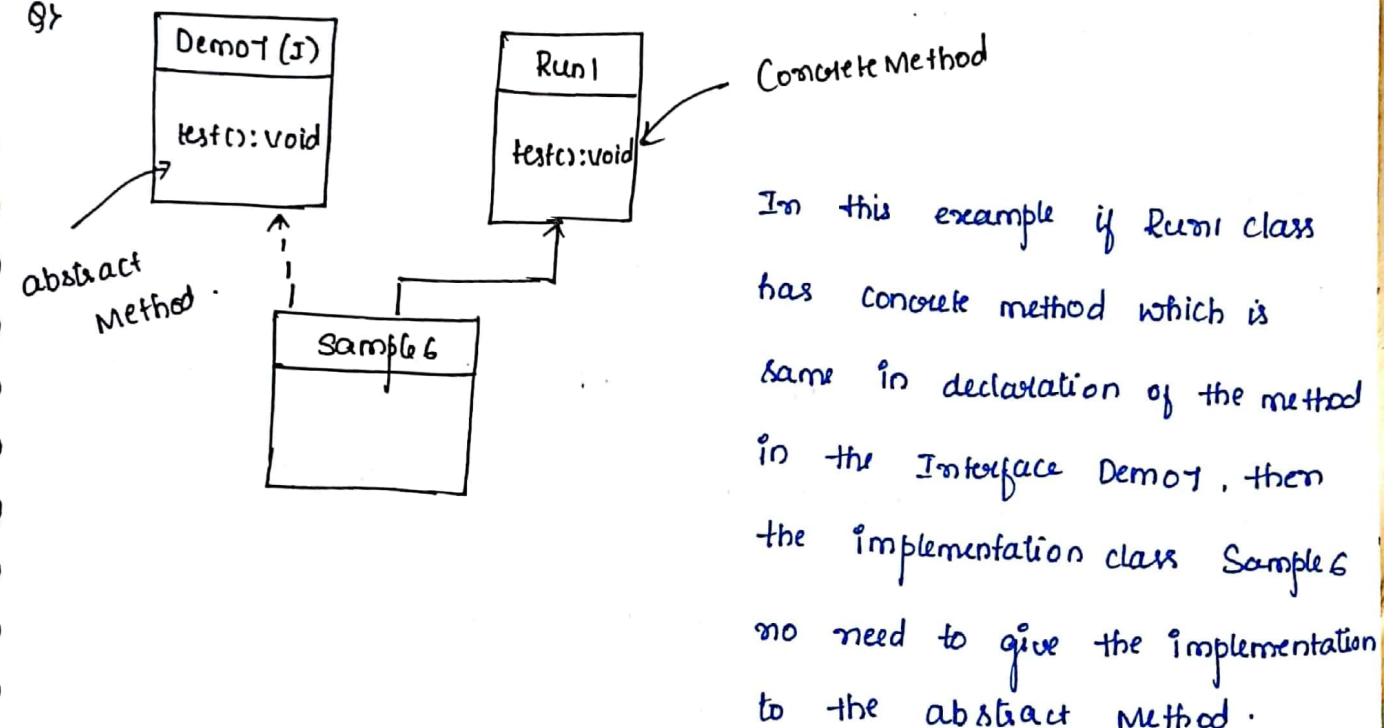
Implementation
class

8)



22/09/2017

9)



Because the concrete method inherit from the class acts as a

implementation for the Interface Method.

→ A class can extend from another class and implements any number of interfaces.

→ An empty interface is known as Marker Interface.

Ex: interface Demo

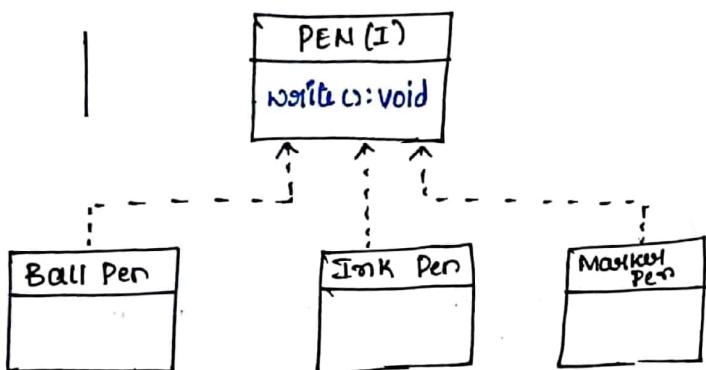
{ }

** ABSTRACTION PRINCIPLE:

"What" are the essential properties? Abstract class OR Java Interface
(functionalities)

"How" these essential properties are implemented? Concrete class
(Multiple Implementation)

"When" these essential properties are used? Creating the object
Accessing the properties



→ Abstraction is one of the OOPS principle which specifies "hiding the implementation of the object functionality".

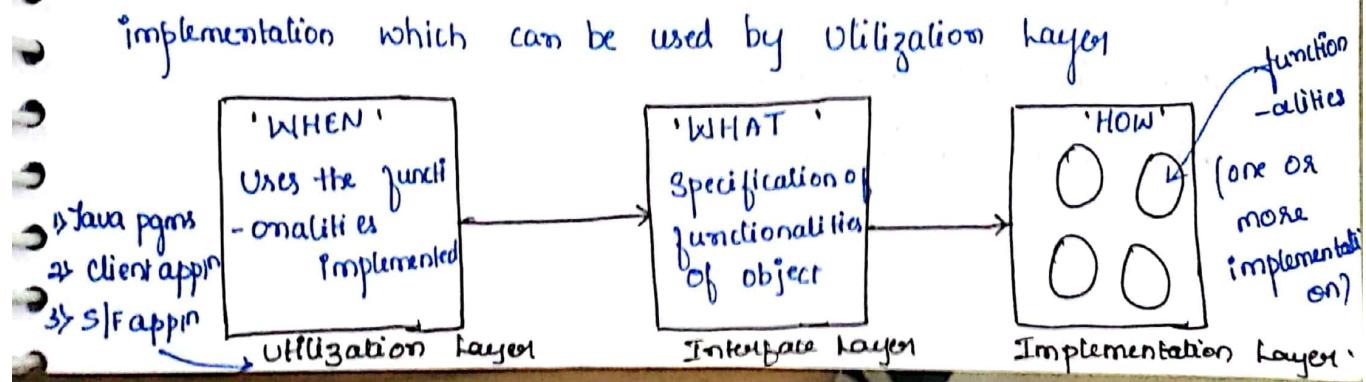
→ In Abstraction the implementation of object, objunctionalities are hidden from the utilization.

→ The utilization part of the program uses the object functionalities without knowing the implementation of the object.

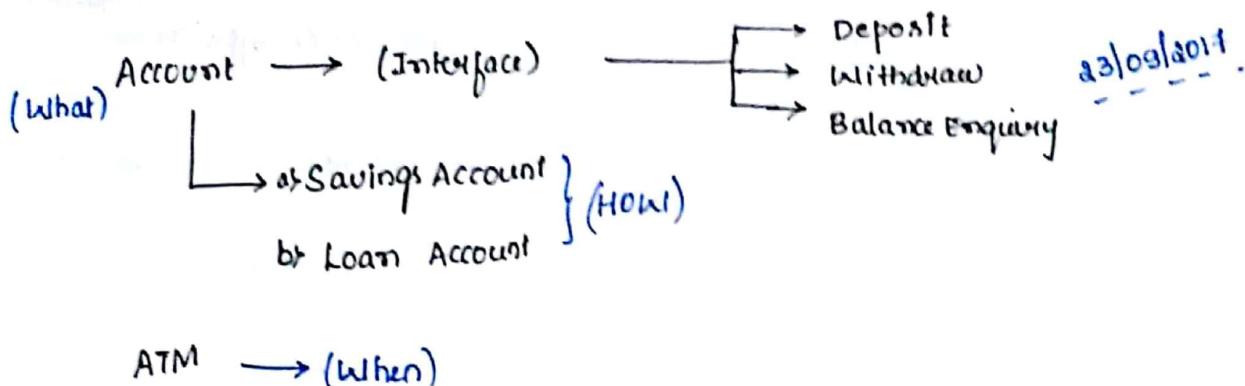
→ The Abstraction code is achieved/written in 3 steps.

- * Declare all the essential properties in a interface Or Abstract class.

- * Write multiple implementation for the interface methods.
- * Access the implementation using Interface ^{type} Reference Variables.
- While designing a Software appn, the Abstraction is achieved by implementing in 3 different layers of Utilization
- at Interface
- 3. Implementation
- The utilization layer uses the functionalities of the object which are present in the Implementation layer.
- This layer interacts with Implementation layer through Interfaces.
- The Interface layer defines all the specifications about the functionalities of the object.
- This is implemented by using Abstract Methods.
- The Implementation layer contains the objects which has multiple implementation and it is binded to the methods at Runtime.
- By using Abstraction any changes made to the implementation will not affect the utilization. We can update or modify or enhance the implementation which can be used by utilization layer.



Example of Abstraction:



Transaction

Savings Account

Loan Account

Initial Balance

10000.00

10,000.00

Deposit 5000

$$\begin{array}{r} + 5,000.00 \\ \hline 15,000.00 \end{array}$$

$$\begin{array}{r} - 5,000.00 \\ \hline 5,000.00 \end{array}$$

Withdraw 7000

$$\begin{array}{r} - 7,000.00 \\ \hline 8,000.00 \end{array}$$

$$\begin{array}{r} + 7,000.00 \\ \hline 12,000.00 \end{array}$$

Account.java

```
package pack1;
```

```
public interface Account
```

```
{
```

```
void deposit(double amt);
```

```
void withdraw(double amt);
```

```
void balanceEnquiry();
```

```
}
```

SavingsAccount.java

```
package pack1;  
public class SavingsAccount implements Account  
{
```

```
    String custName;  
    double accBal;  
  
    public SavingsAccount (String custName, double accBal)  
{
```

```
        Super();  
        this.custName = custName;  
        this.accBal = accBal;  
    }
```

```
    public void deposit (double amt)  
{  
        System.out.println ("Depositing Rs" + amt);  
        accBal = accBal + amt;  
    }
```

```
    public void withdraw (double amt)  
{  
        SOP ("Withdrawing Rs" + amt);  
        accBal = accBal - amt;  
    }
```

```
public void balanceEnquiry ()  
{  
    SOP ("Balance is " + accBal);  
}  
}
```

LoanAccount.java

```
package pack1; public class LoanAccount implements Account  
{  
    String custName;  
    double accBal;  
    public LoanAccount (String custName, double accBal)  
    {  
        this.custName = custName;  
        this.accBal = accBal;  
    }  
    public void deposit (double amt)  
    {  
        SOP ("Depositing Rs" + amt);  
        accBal = accBal - amt;  
    }  
    public void withdraw (double amt)  
    {  
        SOP ("Withdrawing Rs" + amt); accBal = accBal + amt; ?  
    }  
}
```

```

public void balanceEnquiry()
{
    SOP ("outstanding Balance " + accBal);
}
}

```

TestBankApp.java

```

package pack1;

public class TestBankApp
{
    PSVM (String [] args)
    {
        SOP ("Welcome to Global Banking System");

        Account a1 = new SavingAccount ("Ramu", 10000.00);

        a1. balanceEnquiry ();
        a1. deposit (5000.00);
        a1. balanceEnquiry ();
        a1. withdraw (1000.00);
        a1. balanceEnquiry ();
    }
}

```

* *
it can also be done OR create object using Saving Account but we won't achieve Abstraction
we can achieve Polymorphism
* *

Database Calling
 Account a1 =
 AccountDB.getAccount ("Ramu",
 10000.00, 'S');

This is the advantage of ABSTRACTION.

By doing this we can implement many new features like Dmat Account etc. without changing the current features.

Factory Design Pattern

```
package pack1;
```

```
public class AccountDB
```

```
{
```

JAVA DOC

```
/**
```

* the method return an object of type Account based on

* @ param accType

* If accType is 'S' - returns SavingsAccount object

* if accType is 'L' - returns LoanAccount object

* @ param custName

* @ param initAmt

* @ param accType

* @ return Account

```
*
```

```
static Account getAccount (String custName, double initAmt,  
                           char accType) {
```

```
{
```

```
    Account acc = null;
```

```
    if (accType == 'S')
```

```
{
```

```
        acc = new SavingsAccount (custName, initAmt);
```

```
else if (acctType == 'L')
```

```
{
```

```
    acc = new loanAccount (custName, initAmt);
```

```
}
```

```
return acc;
```

```
}
```

```
}
```

Here we give
the defns for
new features

JAVA LIBRARIES

26/09/2017

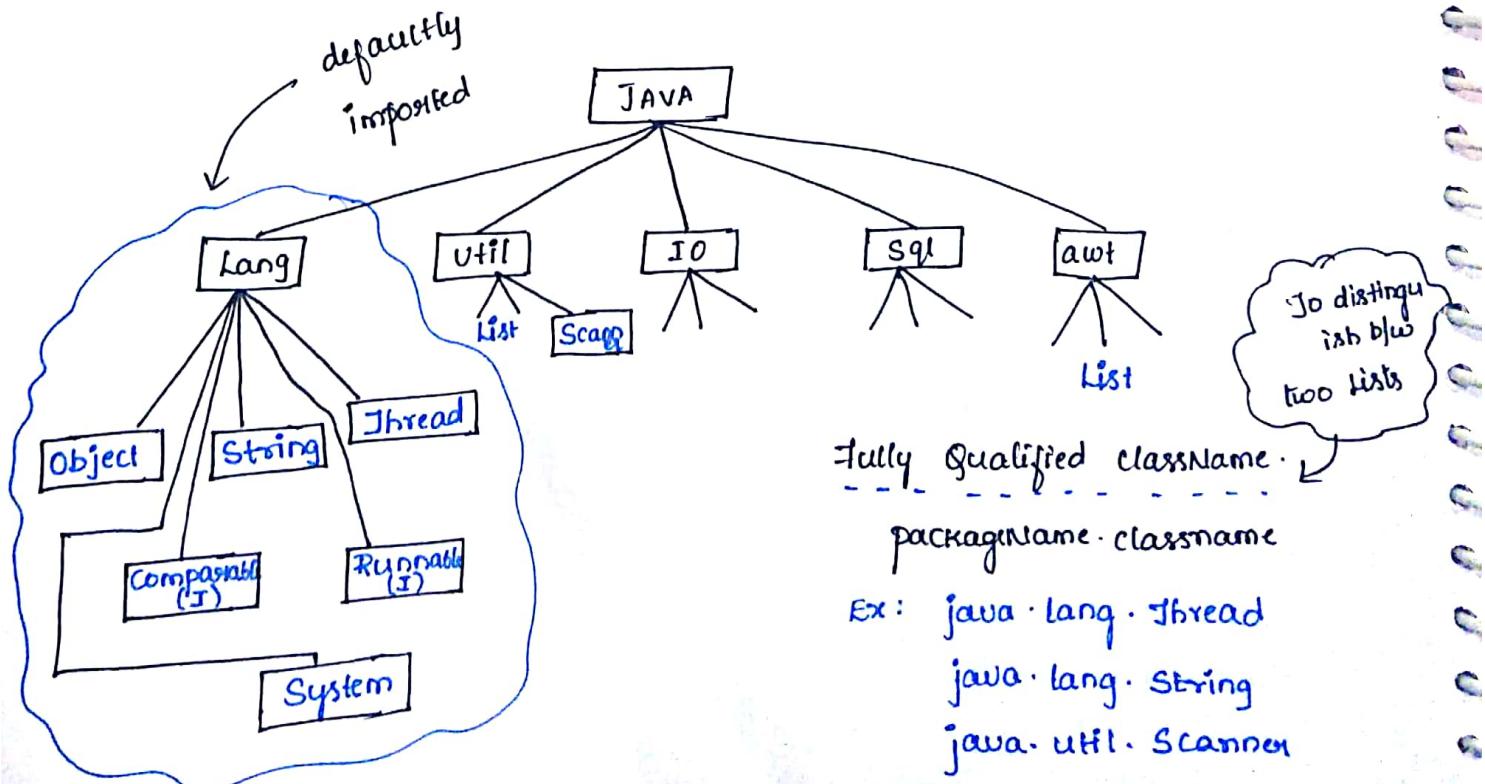
- Collection of Reusable functions.
- Libraries are defined by
 - ↳ Language or Built in Library.
 - ↳ User defined library.
- In java, Library functions are bundled in jar files.

JDK

- provides the necessary library, useful for developing application.

JRE System Library. → contains lot of jar file.

* Important jar file is rt.jar → contains general purpose java library functions
Path: Program Files / Java / jdk1.8.0_151 / lib / rt.jar



** OBJECT CLASS AND ITS

- Object class is a root class of Java library. It is defined in java.lang package.
- It has zero or no argument constructor.
- It has following Non-static functions.

Object Class Functions

- 1) public String toString()
- 2) public int hashCode()
- 3) public boolean equals (Object arg)
- 4) public void wait()
- 5) public void wait (long time)
- 6) public void notify()
- 7) public void notifyAll()
- 8) public void finalize()
- 9) public Object clone()
- 10) public Class getClass()

* public String toString()

→ returns String representation of object.



"fully qualified classname@hexadecimal address"

Ex1 : Object o1 = new Object();

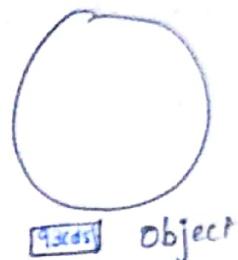
String s1 = o1.toString();

SOP(s1); // java.lang.Object @ 80c3fd

Ex2 : Object o2 = new Object();

String s2 = o2.toString();

SOP(s2); // java.lang.Object @ 93cd5f



To-string Not Overriden in Student class

package pack1;

```
public class Student  
{  
}
```

package pack1;

```
public class MainClass2  
{  
    PSVM (String [] args)  
    {
```

Student st1 = new Student();

String s1 = st1.toString();
 SOP(s1);

O/P:

pack1.Student @ 1db...

pack1.Student @ 1cf....

Student st2 = new Student();

String s2 = st2.toString();
SOP(s2);

}

`toString()` Overrided in the Student class

```
package pack1;  
public class Student  
{  
    public String toString()  
    {  
        //Write your own implementation  
        return "Student instance";  
    }  
}
```

O/P

Student instance

Student instance

- The `toString()` is a member of Object class and it is implemented to return String representation of the Object class.
- It represents fully qualified classname and address of the object in String format.
- On invoking this method on a object, we get a String indicating the classname and the address.
- This method implementation can be changed in the user defined classes if required.

→ Generally in a project while defining a class, we override `toString()` to represent the object states in the String format. **

Example :

```
package pack1;  
public class Student  
{  
    int rollno; String name
```

```

public Student (int rollno , String name)
{
    this . rollno = rollno ;
    this . name = name ;
}

public String toString ()
{
    return "Student [ROLLNO : " + rollno + ", Name : " + name + "]";
}

```

O/P

Student [ROLLNO:145
Name:Ramu]

* public int hashCode()

- This method returns the hashCode number of the object.
- On invoking this method, it returns an integer number which is generated based on the hexadecimal address of the object.
- We can change the implementation of this method by Overriding.

We should implement our own algorithm which generates unique hashCode number.

// Main class Program

Ex: int n1 = St1 . hashCode ();

O/P

int n2 = St2 . hashCode ();

n1 = 12345

n2 = 64321

SOP (n1);

SOP (n2);

}

Different values

** EQUALS() Method : public boolean equals (Object arg)

int x = 10;

int y = 10;

x == y

Value of x is equal to value of y

Same type

Demol d1 = new Demol();

Demol d2 = new Demol();

d1 == d2

X value of d1 is equal to value of d2

equals() ✓

d1.equals(d2);

** It checks object address object reference.

** It checks object value
based on 'HashCode'

d1.equals(d2)

Current object

given object

Ex: package pack1;

public class Demol

{

int x;

public Demol (int x)

{

this.x = x;

}

public boolean equals (Object arg)

{

Demol ref = (Demol) arg;

return this.x == ref.x;

}

public String ^(caps) toString ()

Here object properties are exhibited.

So, Downcast to show the Demol properties.

return "Demol [x=" + x + "]";

```
package pack1;  
public class Mainclass  
{  
    public static void main (String [] args)
```

```
    {  
        Demol d1 = new Demol(25);
```

O/P

```
        Demol d2 = new Demol(25);
```

```
        SOP (d1);
```

```
        SOP (d2);
```

```
        if (d1.equals (d2))
```

→ Objects are not same in
property x before overriding
the equals (object arg)

→ After overriding, we get

```
    {  
        SOP ("object are in the same property");  
        object are in the  
        same property.
```

```
}
```

```
    else {  
        SOP ("objects are not same in property x");
```

```
}
```

```
}  
-----  
Example: Using Multiple Values.
```

```
package pack1;
```

```
public class Demol
```

```
{  
    int x;
```

```
    double y;
```

```
public Demol (int x, double y)
```

```
{  
    this.x = x;  
    this.y = y;  
}
```

```
public boolean equals (object arg)
{
    Demo1 ref = (Demo1) arg;
    return this.x == ref.x && this.y == ref.y;
}

public String toString ()
{
    return "Demo1 [x=" + x + ",y=" + y + "]";
}

package pack1;

public class Mainclass
{
    public void (String [] args)
    {
        Demo1 d1 = new Demo1 (25, 4.3);
        Demo1 d2 = new Demo1 (22, 4.3); if (d1.equals (d2)) {
            SOP ("object are same in property x and y");
        }
        else {
            SOP ("object are not same in property x & y");
        }
    }
}
```

- Equals() of object class is used to check equality between same type of object.
 - In object class it is implemented to check the equality of object on the basis of hashCode number of the object.
 - It returns true, if the current object hashCode is equal to the given object hashCode.
 - If it is not equal in hashCode, it returns false.
 - If we have to check two objects are equal in the property then we should override equals().
 - We can override equals() to check the equality either on one property or multiple property.
 - Whenever we use Set type of collections in project equals() & hashCode() are overridden.
-

Q) Write a java program to check the time on one wall clock is equal to the time on another wall clock. If both clocks are showing same time, the pgm should display the o/p the clock time are same otherwise clock's time are not same.

*
NOTE: In java lang, whenever we print a reference variable, it internally call the toString() of the object, the print statement will print the string returned from the toString().

If `toString()` is not overridden, then we get object class implementation. If `toString()` is overridden, we get Overridden Implementation.

28/09/2017

** STRING CLASS:

- String classes are immutable.
- String class is Thread Safe.
- Implements Comparable Interface. Hence String objects are mutually comparable in nature.
- String class has overloaded constructors.
- The String class object will be created in two ways:
 - ↳ Using New operator.
 - ↳ Without new operator.
- The strings created without new operator are known as String literals.
- String objects are stored in Separate memory area known as String Pool.
- The String pool is divided into two areas
 - ↳ String Constant Pool
 - ↳ String Non-Constant Pool

Area

- String constant pool area does not allow duplicate string whereas Non-constant pool area allows duplicate string.
- A String created using new operator will always reside in Non-constant Pool area.
- In String class the following methods are Overriden -
 - * toString() : toString is Overriden to return the String value stored in the object.
 - * hashCode() : hashCode is overriden to generate hashCode number based on String value not on object address.
 - * equals() : is overriden to check the equality of strings based on string value.

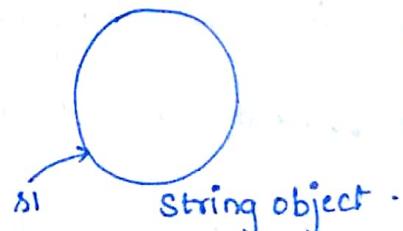
** CONSTRUCTORS OF STRING CLASS

The String class has overloaded constructors, few of the constructors are :

(i) NO Arg Constructor

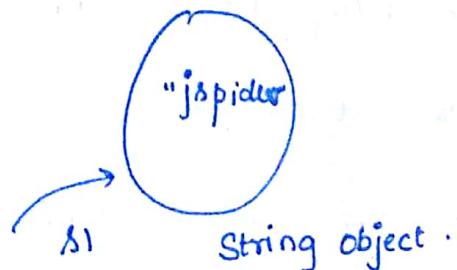
```
String s1 = new String();
```

↳ Creating empty object.



ii) String arg type Constructor

```
String s1 = new String ("jspider");
```

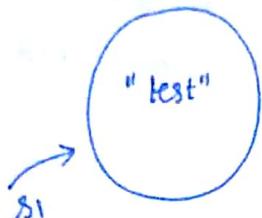


3) char Array type constructor :

```
char[] ch = { 't', 'e', 's', 't' };
```

```
String s1 = new String (ch);
```

create a String object with the char array element.



Converting char Array type to String type.

Example:

```
package pack1;
```

```
public class Demo1
```

```
{
```

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

```
{
```

```
    String s1 = new String ("jspiders");
```

```
    String s2 = new String ("jspiders");
```

```
    SOP ("s1=" + s1);
```

```
    SOP ("s2=" + s2);
```

```
    int n1 = s1.hashCode ();
```

```
    int n2 = s2.hashCode ();
```

```
    SOP (n1);
```

```
    SOP (n2);
```

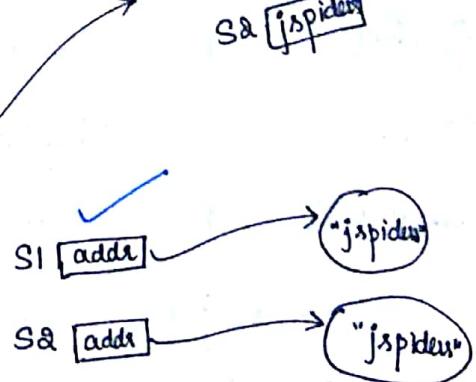
```
    if (s1 == s2)
```

```
    { SOP ("two strings are same"); }
```

address of object is referred to s1

X
S1 [jspider]

S2 [jspider]



```

else
{
    SOP ("two strings are not same");
}

```

O/P

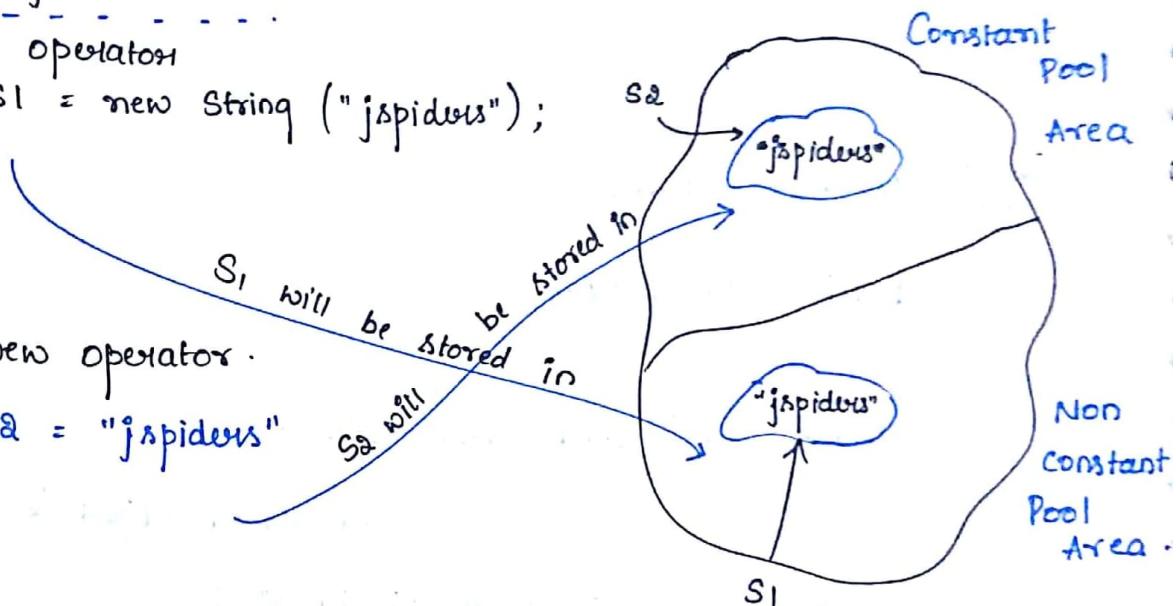
two strings are not same

if we use equals() then
two strings are same.

* String Object Creation.

Using new operator

1) String s1 = new String ("jspiders");



2) Without new operator.

String s2 = "jspiders"

Important truly questions:

String s1 = new String ("jspiders");

String s2 = new String ("jspiders");

String s3 = "jspiders";

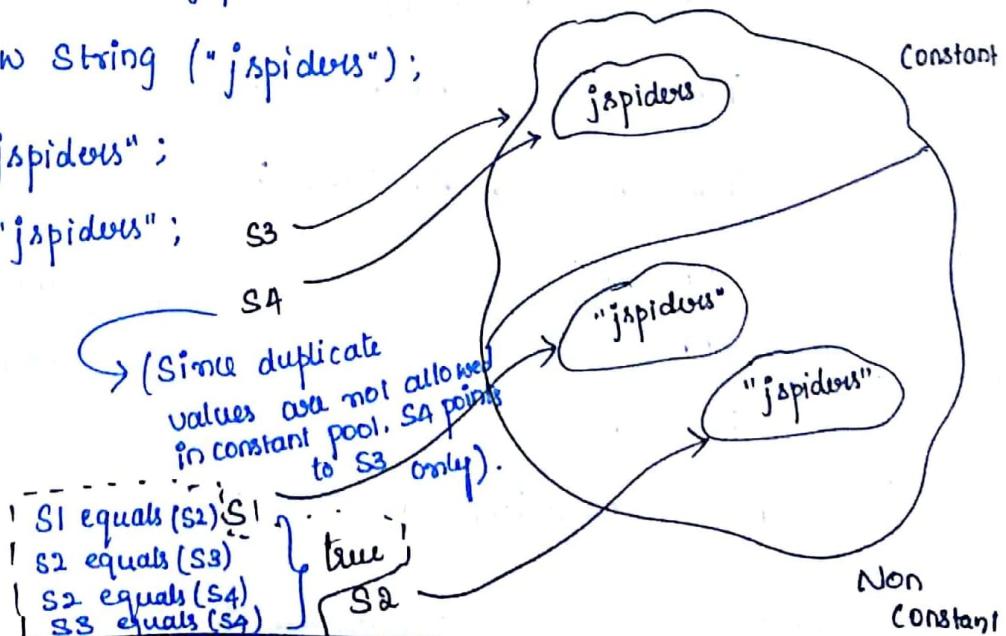
String s4 = "jspiders";

s1 == s2 false

s2 == s3 ; false

s2 == s4 ; false

s3 == s4 ; true



2*) Important truly question

```
String s1 = "java";
```

If we concat two literals then the result will be literal.

```
String s2 = "developer";
```

One literal If one " " are present then one object will be created

```
String s4 = "java"+ "developer"; → literal.
```

```
String s5 = "java" + s2; → Not a literal
```

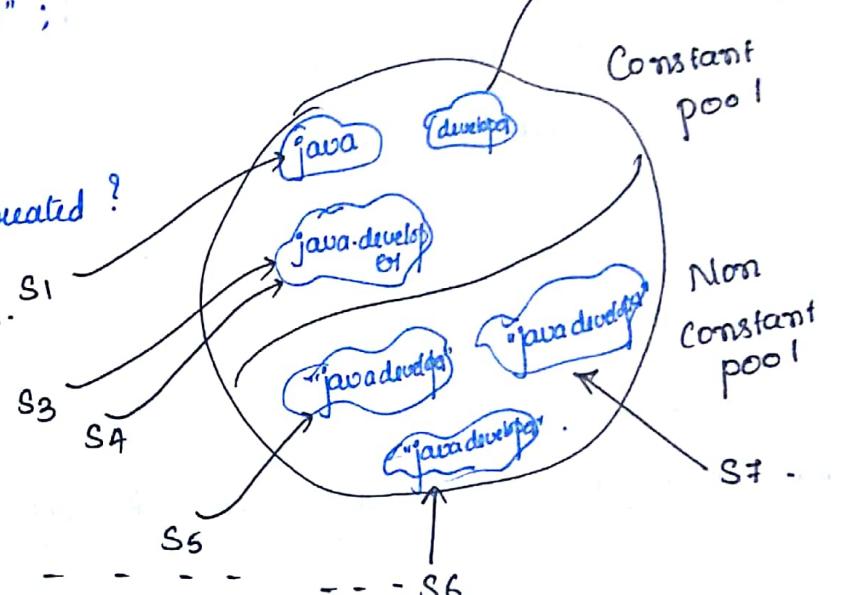
It create the new object using new operator

```
String s6 = s1 + "developer";
```

```
String s7 = s1 + s2;
```

How many objects are created?

6 objects with 7 references.



```
String s1 = new ("Ramesh");
```

```
String s2 = new String ("Surush");
```

3 objects are created.

```
String s3 = new String ("Umesh");
```

```
String s1 = "Ramesh";
```

Ramesh

```
String s1 = "Surush";
```

Ramesh

```
String s1 = "Umesh";
```

Surush

We cannot change the object value. This is known as Immutable.

Finally, Reinitialization.

Umesh

In java only we can reinitialize only Reference variable.

```
String s1 =  
    "java"
```

Pgm1.java

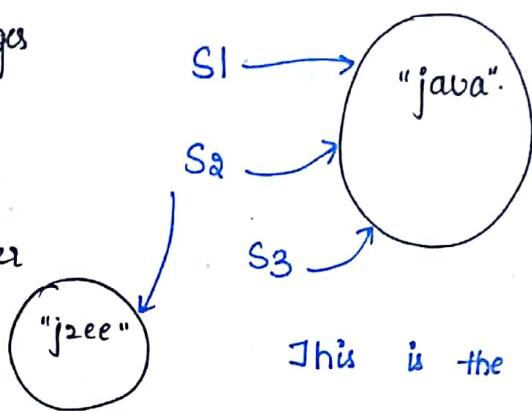
```
String s1 =  
    "java"
```

Pgm2.java

```
String s1 =  
    "java"
```

Pgm3.java

When pgm2 changes
the value to "j2ee"
then s2 refers
to another
Object



This is the advantage of
Immutable.

** STRING BUILDER AND STRING BUFFER.

03/01/2017

- Both the classes are defined in java.lang packages.
- Both are used to store String values.
- Both are mutable objects. (we can change the existing values).
- Both are final class.
- In both the classes only `toString()` is overridden.
- `HashCode` & `equals()` are not overridden.
- Both classes does not implement Comparable Interfaces, both are not mutually Comparable.
- The array of both the objects cannot be sorted.

→ Both classes have same method and perform same operation, the only difference between them are StringBuffer class is a Thread Safe whereas StringBuilder is not a Thread Safe:

→ These class objects are created using new operator only.

Example:

```
package pack1;  
public class Demo2  
{  
    public static void main (String [] args)
```

```
    StringBuilder sb1 = new StringBuilder ("developer");
```

```
    SOP ("given string is " + sb1); // Functions defined in String
```

```
    SOP ("appending character");
```

```
    sb1.append ('s');
```

It is a overloaded Method

```
    SOP (sb1);
```

it accepts numeric datatype
as well as String .

```
    SOP ("Inserting a character");
```

```
    sb1.insert (2, 'w'); // new objects are not created to
```

```
    SOP (sb1);
```

change the values. Because these are

```
    SOP ("Replace a character");
```

Mutable

```
    sb1.setCharAt (5, 'L');
```

```
    SOP (sb1);
```

```
}
```

→ SOP ("Reversing the String");

```
    sb1.reverse ();
```

```
    SOP (sb1);
```

Example :-

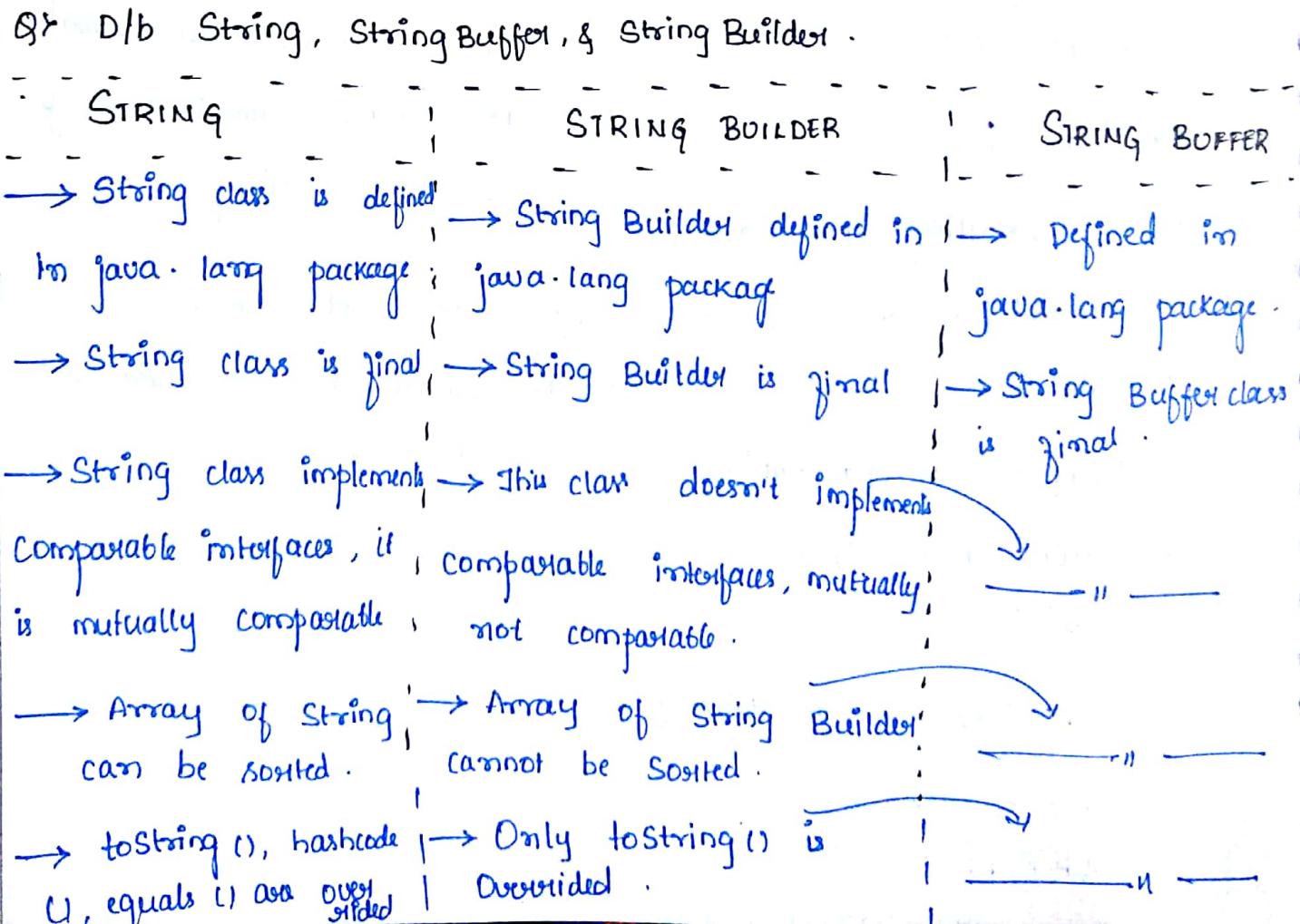
```
package pack1;

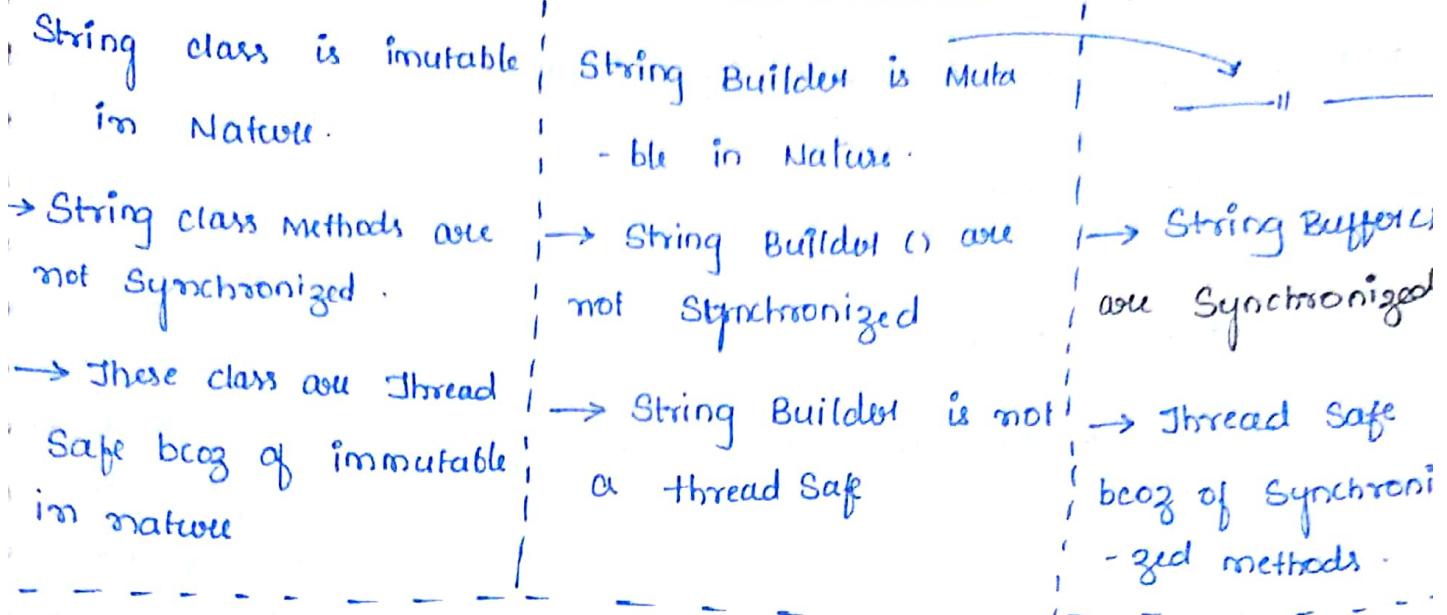
public class Demo3

{
    public static void main (String [] args)
    {
        String s1 = "developer";
        // represent the String type in StringBuilder type.
        StringBuilder sb1 = new StringBuilder(s1);
        // do all manipulation.

        s1 = sb1.toString();
    }
}
```

→ D/b String, StringBuffer, & StringBuilder.





** HAS-A RELATIONSHIP :

```

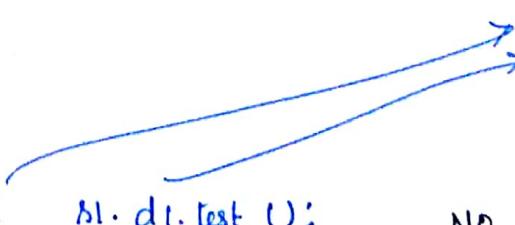
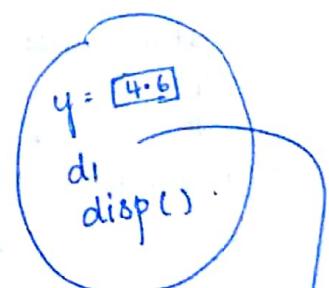
class Demo1
{
  int x=10;
  void test()
  {
    SOP ("Running test()");
  }
}
  
```

```

class Sample1
{
  double y = 4.6;
  Demo1 d1 = new Demo1();
  void disp ()
  {
    SOP ("running disp ()");
  }
}
  
```

```

class Mainclass
{
  PSVM (String [] args)
  {
    Sample1 s1 = new Sample1();
    SOP (s1.y);
    s1.disp ();
    SOP (s1.d1);
    SOP (s1.d1.x);
  }
}
  
```



No error or O/P is displayed.

Example 2 : Example for Static Reference Variable.

```
class Sample  
{  
    double y = 4.6;  
  
    static Demo1 d1 = new Demo1();  
  
    void disp()  
    {  
        SOP ("running disp");  
    }  
}
```

In Main Method

Real-Time Examples:

Mobile $\xrightarrow{\text{Has-A}}$ Battery

Mobile $\xrightarrow{\text{Use-A}}$ Sim

SOP (Sample1.d1);

SOP (Sample1.d1.x);

Sample1.d1.test();

ClassName Static
Ref. Variable

We cannot create
Multiple object
since we use
Static Keyword.

System.out.println();

ClassName Static ref variable
of Print Stream

Non Static Method (Overloaded)

System.in.read();

Standard I/O device.

** ARRAYS:

arraytype [] arrayname = new arraytype [size];

Types of Arrays:

* Array of primitive type

int [] a1 = new int[5];

hold 5 int values

* Array of Non-primitive type

Demol [] a1 = new Demol[5];

hold 5 objects of Demol class

Examples : Student.java

```
package pack1;
public class Student {
    int rollno;
    String name;
    double marks;
    public Student (int rollno, String name, double marks) {
        this.rollno = rollno;
        this.name = name;
        this.marks = marks;
    }
    public String toString() {
    }
}
```

```

return " Student [ rollno = " + rollno + ", name = " + name + ", marks
        +
        + marks + " ] ";
}

package pack1 / Mainclass

public class Demo1

{
    PSvm (String [] args)
    {
        Student [] stArr = new Student [5] ;

        // Storing Student Details in Array.

        stArr [0] = new Student (124, "Ramesh", 65.78);
        stArr [1] = new Student (125, "Swadesh", 75.78);
        stArr [2] = new Student (123, "Umesh", 80.28);
        stArr [3] = new Student (121, "Kalluhi", 81.28);
        stArr [4] = new Student (120, "Jagguhi", 80.28);

        SOP (" Total students : " + stArr.length);

        SOP (" Display all student Details");

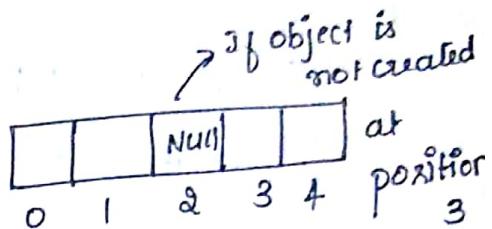
        SOP (" RollNO \tName \tMarks");

        for (int i=0; i<stArr.length; i++)
        {
            SOP ( stArr [i].rollno + "\t" + stArr [i].name + "\t" +
                  stArr [i].Marks);
        }
    }
}

```

//Reusable code. (NoticeBoard.java)

```
package pack1;  
public class NoticeBoard  
{  
    void display (Student [] arr)  
{  
        SOP (" RollNo | Name | Marks");  
        for (int i=0; i< arr.length; i++) {  
            if (arr[i] != null)  
                SOP (arr[i]. rollno + " | " + arr[i]. name + " | " + arr[i].  
                      Marks);  
        }  
    }  
}
```



↓
Null Pointer Exception
will occur so

Avoid this make
use of ↗

Ex: psvm (String [] args)

```
{  
    Student [] stArr = new Student [5]
```

// Storing Student Details in Array .

stArr [0]

stArr [1]

stArr [3]

stArr [4]

SOP ("Total Students: " + stArr.length);

SOP ("Display all student Details");

```
NoticeBoard b1 = new NoticeBoard (); // To access  
b1.display (stArr); Nonstatic method.
```

? }

// To count exact no. of students present except Null values.

```
int studentCount (Student arr)  
{  
    int count = 0;  
  
    for (int i=0; i<arr.length; i++)  
    {  
        if (arr[i] != null)  
            count++;  
    }  
    return count;  
}
```

Length always
gives the capacity
of array not the
Actual Number
of object

// In Main Method calling studentCount();

```
** int count = bl.studentCount (stArr);  
SOP ("Total Students :" + count);  
**
```

Example showing usage of Array : ^{OBJECT}

```
package pack1;  
public class Employee  
{  
    int id;  
    String name;  
    double salary;  
  
    public Employee (int id, String name, double salary)
```

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

public String toString()
{
    return "Employee [id = " + ..... ]";
}
}
```

Sample 1 . java

package pack1;

```
public class Sample1
```

{

```
void updateGraceMarks (Object [] arg, int grace)
```

f

```
for (int i=0; i<arg.length; i++)
```

9

if (arg[i] != null)

1

if (arg[i] instanceof Student)

{

Student 81 = (Student) arg [i];

81. marks = 81. marks + grade;

}

```
void displayStudentDetails (Object [] arr)
```

f

SOP (" ROLLING \tName \tMarks ") ;

```
for (int i=0; i<array.length; i++)
```

```

if (args[1] != null)
{
    if (args[1] instanceof Student)
    {
        Student s1 = (Student) args[1];
        SOP(s1.rollno + " | " + s1.name + " | " + s1.marks);
    }
}

```

Demos.java.

```

public class Demos
{
    PSVM (String [] args)
    {
        Object [] objArr = new Object [6];
        objArr[0] = new Student (124, "Ramesh", 65.78);
        objArr[1] = new Employee (12345, "Swadesh", 25000.00);
        objArr[2] = new Student (
        objArr[3] = new Employee (
        objArr[4] = new Employee (
        objArr[5] = new Student (
            Sample s1 = new Sample ();
            s1.displayStudentDetails (objArr);
            SOP ("Updating grace Marks to Students");
            s1.updateGraceMarks (objArr, 5);
            s1.displayStudentDetails (objArr);
        }
    }
}

```

Heterogeneous type of objects

** LIMITATIONS OF ARRAYS:

- The array has fixed size, it cannot grow dynamically.
- Array generally supports same type of elements. we cannot store different types of elements in array.
- Array doesn't support any standard data structure. It always stores elements linearly.
- Array doesn't provide any imbuilts algorithms for searching & sorting.

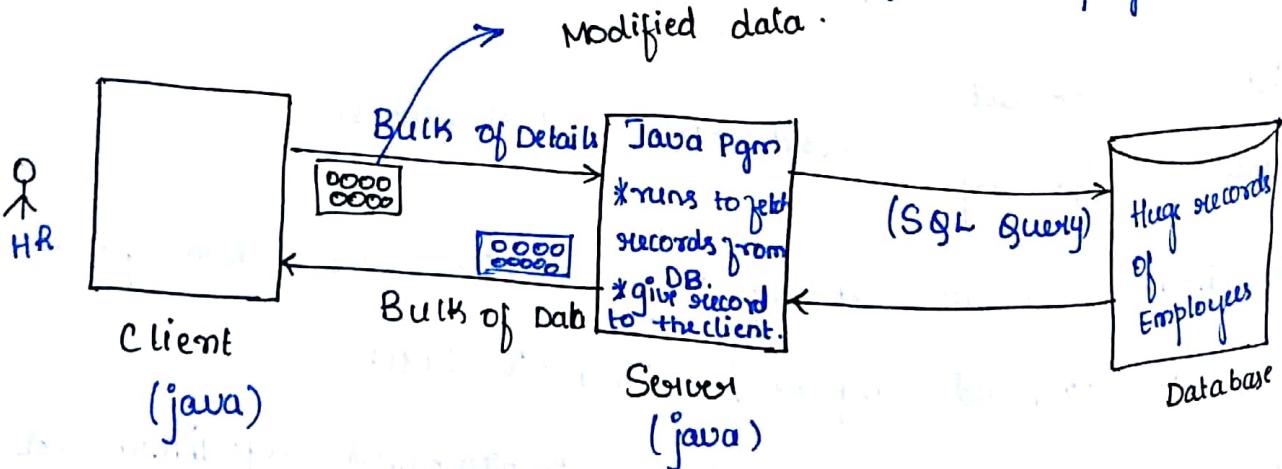
- * Arrays are preferable when we know exact no. of elements, type of elements & to store elements linearly.
- If we don't know the no. of elements & type of elements, we want to store the elements non-linearly, then we go for

DATA STRUCTURES: (In java we call it has 'COLLECTIONS').

** INTRODUCTION TO COLLECTIONS

05/10/2017

Ex: Use Case: Increment 5% salary of all Employees.

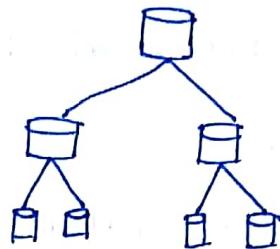


** COLLECTION FRAMEWORK LIBRARY:

1) Music Player → Line up songs → Queue.

2) Mail Box → Latest will be up → Stack.

3) File System → Tree.



→ Representing a group of objects as a single entity is known as Collection.

→ The object must be of same type or different type.

→ The collection does not have any limitation to the size, it grows dynamically.
Ex : * A group of song in a playlist is a collection
* A group of mails in an inbox is a collection
* A group of apps in an Android or Iphone is a collection.

→ Collection Framework Library is a set of libraries interfaces and classes which implements the standard data structures like :

1) List

3) Set

and other data structures.

2) Queue

4) Map

→ The library provides set of implementation class where program make use of it to build required Data Structures.

→ The library also provide set of implemented algorithm for

accuring the elements for sorting and searching.

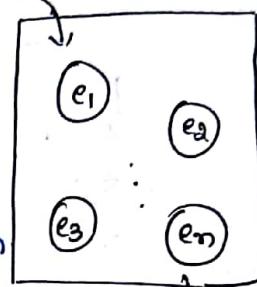
→ The benefit of this libraries are : 1) Accurate & Robust

3) Easy to use & 4) Customizable & 5) Portable.

* Fundamental Of a Collection :

* Add Operation:

Add



→ Adding an element (object) into the collection

→ Element is created to object class type collection.
(Upcasting)

↓ Retrieve Or Remove

→ Elements present inside collection shows Object class properties.

* Retrieve Operation:

→ getting access to the element present inside collection.

→ Downcast to get original properties.

* Remove operation:

→ removing the element from the collection.

→ removed object shows object class properties.

→ need downcasting get original properties.

** TYPES OF COLLECTIONS:

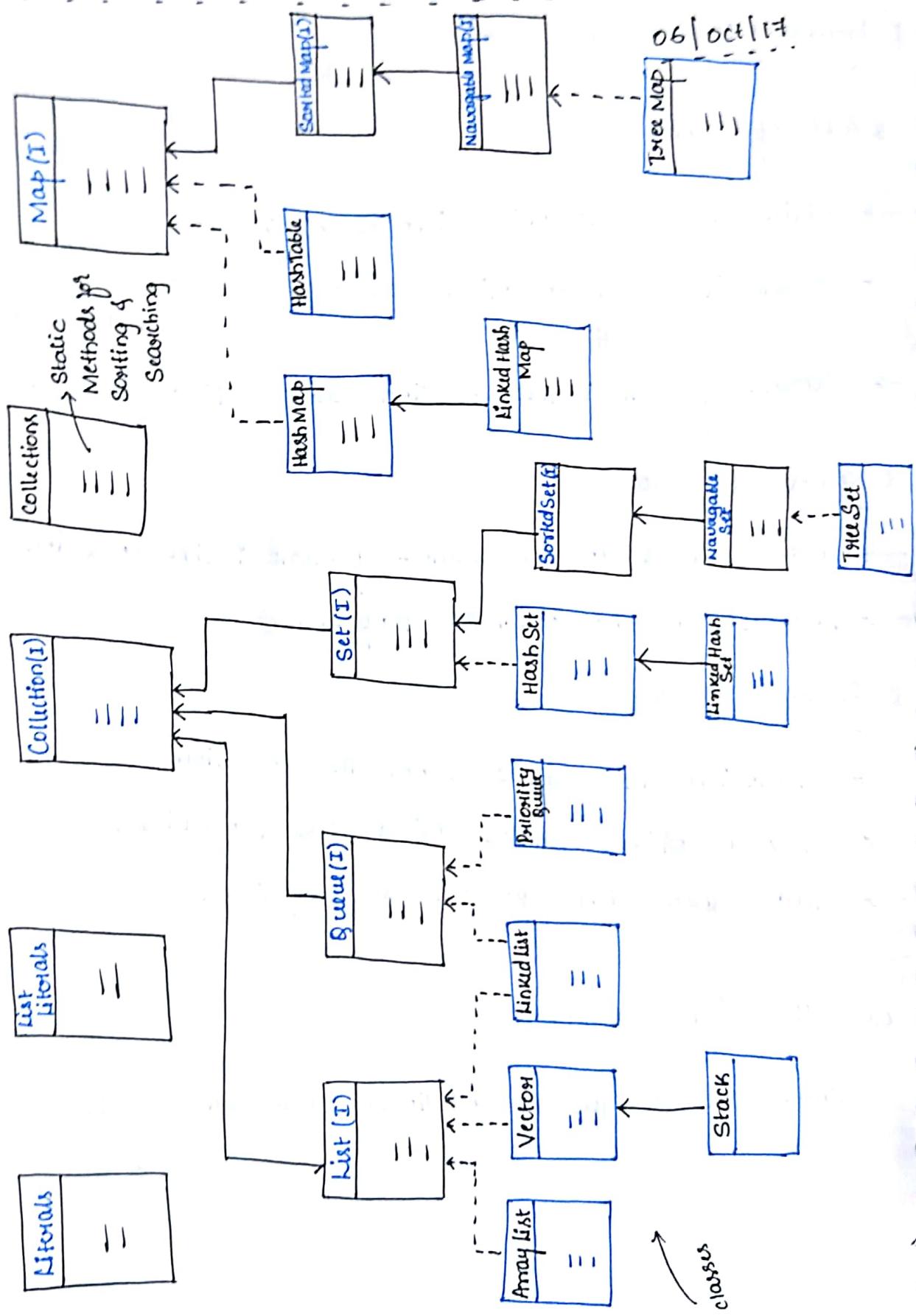
There are 3 types of collections. They are : 1) List

2) Queue

3) Set

- List stores elements with index.
- Queue stores elements without index but processed in FIFO.
- Set stores elements without index but no duplicates in Set.

Java Collection Framework Hierarchy



Collection is the root interface of collection framework library hierarchy.

→ It specifies all the common operations of the collection types which must be implemented by all the implementation classes.

→ The methods of collection Interface are:

1) public boolean add (Object e)

The method inserts the specified elements into the collection.

The element can be any type of object, if the element is added successfully it returns true otherwise it returns false.

2) public int size ()

The method returns numbers of elements present in collection.

3) public boolean isEmpty ()

On invoking this method it returns true if the collection is empty otherwise the method returns false if the collection is not empty.

4) public void clear ()

→ On invoking this method, it makes collection empty by removing all elements present in the collection.

5) public boolean contains (Object e)

The method is used to find the specified element is present in collection or not.

It returns true if element present in collection else it returns false.

6) public boolean remove (Object e)

→ The method removes the specified element from the collection. If the element is removed successfully, it returns true else it returns false.

7) public Iterator iterator()

→ The method returns an object of Iterator type.

8) public Object[] toArray()

→ The method returns the array of Object which contains the elements of collection.

In other words, this method converts the collection into array of object types.

9) public boolean addAll (collection c)

The method is used to add all the elements specified in the argument into the current collection.

If all elements are added successfully it returns true else false.

10) public boolean containsAll (collection c)

The method returns true if all the elements present in the given collection is found in the current collection, if not returns false.

11) public boolean removeAll (collection c)

The method returns true after removing common elements from the current collection otherwise it returns false.

- In other words, the method removes all the elements in the current collection which are also found in the given collection.
- ↳ If public boolean retainAll(Collection c)
- After invoking this method, the current collection will have only common elements, the uncommon elements are removed from the current collections.

Example 1) Let's say $c_1 \rightarrow$ collection containing e_1, e_2, e_3, e_4
 $c_2 \rightarrow$ collection containing f_1, f_2, f_3 .

$c_1.addAll(c_2)$

$c_1 \rightarrow e_1, e_2, e_3, e_4, f_1, f_2, f_3$.

2) Let's say $c_1 \rightarrow$ collection containing e_1, e_2, e_3, e_4
 $c_2 \rightarrow$ collection containing e_2, e_3 .

$c_1.containsAll(c_2) \dashrightarrow$ return is true.

3) Let's say $c_1 \rightarrow$ collection containing e_1, e_2, e_3, e_4
 $c_2 \rightarrow$ collection containing f_1, f_2, e_2, f_3, e_4

$c_1.removeAll(c_2)$

$c_1 \dashrightarrow e_1, e_3$.

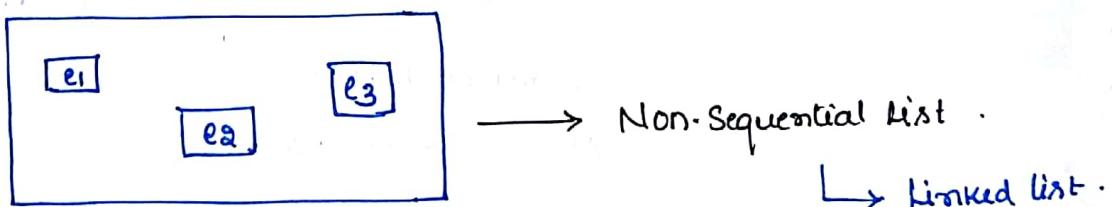
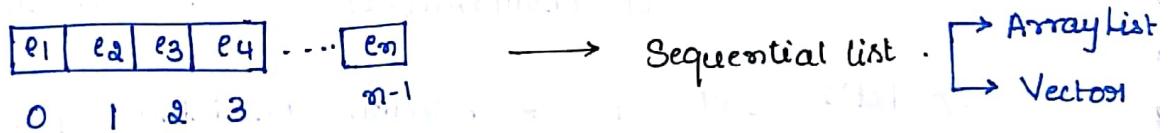
4) Let's say $c_1 \rightarrow$ collection containing e_1, e_2, e_3, e_4
 $c_2 \rightarrow$ collection containing f_1, f_2, f_3, e_2

$c_1.containsAll(c_2)$

$c_1 \rightarrow c_2, e_4$.

* LIST

- List is a type of collection which stores elements with index.
- It is known as Indexed Collection.
- List allows duplicate element.
- List allows Null Insertion.
- List preserves the order of insertion until unless it is not disturbed.
- On any list, we can perform operations based on Index.
- List elements are stored either Sequentially or Non-Sequentially.
 - * In case of Sequential list, the elements are stored one after the other whereas
 - * In case of Non-Sequential list, the elements are stored randomly.



Index based oprn.

- 1) remove element @ a index
- 2) Insert element @ an index
- 3) Replace element @ an index
- 4) Retrieve element @ an index

** LIST INTERFACE

- The List Interface is a child Interface of Collection Interface.
- It specifies the general operations which can be performed on any types of lists.
- The list specifies the below methods to describe the list feature
- 1) public boolean add (int index, Object e)
 - The method inserts a specified element @ the specified index in the list.
 - The method returns true if the insertion is success, otherwise it returns false.
 - If already element is present at the specified index, it will be shifted to right which results in shifting all the elements to the right.
 - If list is not having specified index it will throw IndexOut of Bound Exception.
- 2) public Object get (index)
 - The method returns the reference to the element present @ the index.
 - The returned reference type will be Object class type.
 - If index is not found, the method throws ^{out of} Indexbound Exception.
- 3) public Object remove (int index)
 - The method removes the specified element present @ the specified index.
 - The removed element shows object class properties.
 - If the index is not found, the method throws Index Out of Bound Exception.
 - After removing the element, the list element will be shifted to left.

4) public Object set (int index, Object e)

→ The method replaces the element found at the specified index by the element which is specified in and argument.

Ex: set (3, ent)

→ Method returns Original element which was present in the index.

5) public int indexOf (Object e)

→ The method returns the first occurrence of the specified element in the list.

→ If the element is not found, it returns -1.

6) public int lastIndexOf (Object e)

→ The method returns the last occurrence of the specified object

→ If object is not found, it returns -1.

7) public List sublist (int startIndex, int endIndex)

→ The method returns the part of the list from the given list between the indices specified in the arguments.

→ The end index will be excluded.

8) public ListIterator listIterator ()

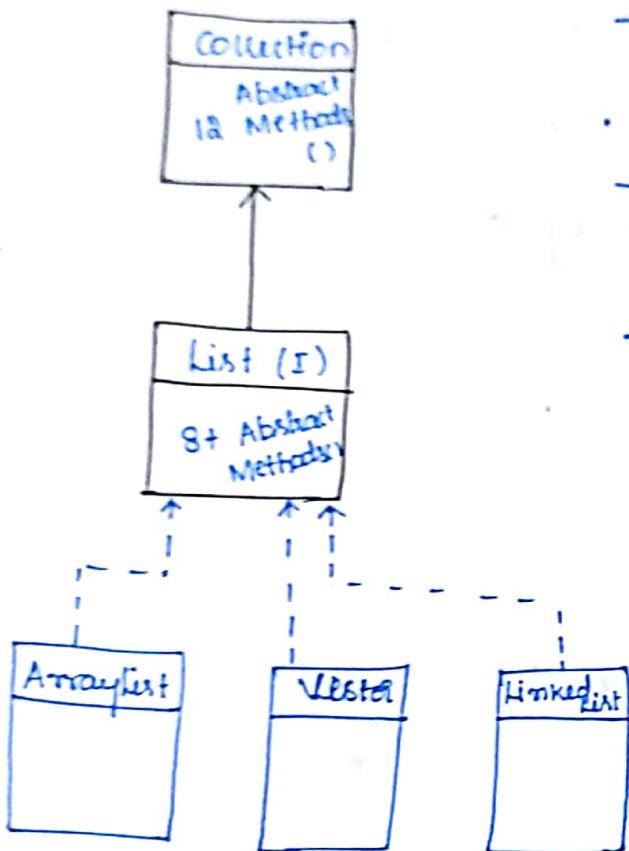
→ The method returns object of ListIterator type

* If we point reference variable to collection it returns the value

* String is an object in JAVA.

* Wrapping or Automatic Boxing introduced from jdk 1.5

* Implementation classes of List Interface (List I)



- ArrayList & vector → It provides implementation to store elements sequentially.
- LinkedList - It provides implementation to store elements non-sequentially.
- Vector is a legacy class (old one it was introduced from jdk 1.1 till now no modifications).

Ex: ArrayList l1 = new ArrayList();

List l1 = new ArrayList();
// Upcasting.

Collection c1 = new ArrayList();
-----;

Examples:

```
package listexamples
```

```
import java.util.ArrayList;
```

List l1 = new LinkedList();

```
public class ListDemo1
```

```
{
```

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

```
{
```

```
    ArrayList l1 = new ArrayList();
```

```
// Creates an empty list.
```

OR

LinkedList l1 = new
LinkedList();

{ Same Methods
but different
implementations }

```
SOP("Is list Empty: " + l1.isEmpty());
```

```
SOP("Total list elements: " + l1.size());
```

```
SOP("adding elements into list");
```

```

        l1.add("java");    l1.add("jace"); l1.add("android");
        l1.add(null);     l1.add("java");   l1.add(12);
        System.out.println("Total list elements :" + l1.size());
        System.out.println("Is list empty :" + l1.isEmpty());
        System.out.println(l1);
        System.out.println("Inserting element @ index 1");
        l1.add(1, "angularjs");
        System.out.println(l1);
        System.out.println("Total list elements :" + l1.size());
        System.out.println("Removing element");
        l1.remove("jace");
        System.out.println(l1);
        System.out.println("Total list elements :" + l1.size());
    }
}

```

Example : To retrieve all the elements of ArrayList

```

package listexamples;
import java.util.ArrayList;
import java.util.List;
public class ListDemo {
    public static void main(String[] args) {
    }
}

```

```

List li = new ArrayList();
SOP ("Is list empty" + li.isEmpty());
SOP ("List Size" + li.size());
SOP ("adding elements into list");
li.add ("java"); li.add ("jace"); li.add ("android");
li.add (null); li.add ("java"); li.add (12);
SOP ("element @ index 1:" + li.get(1));
SOP ("list elements");
for (int i = 0; i < li.size(); i++) {
    SOP (i + "-->" + li.get(i));
}
SOP ("Total list elements:" + li.size()); SOP (li);
}

```

* ARRAYLIST

- ArrayList is a implementation class of List Interface.
- ArrayList implements 3 Marker Interface if Serializable
if Cloneable.
if Random Access.
- ArrayList is implemented with a DS resizable array D.S
- ArrayList has a default capacity of 10.
- The ArrayList grows with formula (current capacity * $\frac{3}{2}$) + 1.

→ ArrayList is a sequential type of list.

* Constructors of ArrayList.

→ ArrayList has overloaded constructors.

1) No arg constructor

ArrayList l1 = new ArrayList();

→ Creates an empty list of capacity 10

2) int - arg Constructor

ArrayList l1 = new ArrayList(n);

n → int nos specifies initial capacity.

Ex: ArrayList l1 = new ArrayList(100);

→ Creates an empty list with initial capacity of 100.

3) Collection arg type constructor.

Let's say C1 is a collection of 'n' elements.

ArrayList l1 = new ArrayList(C1);

** Internal Implementation: Creates a list with the elements of given collection type

Example: ArrayList l1 = new ArrayList()

l1.add(e1)

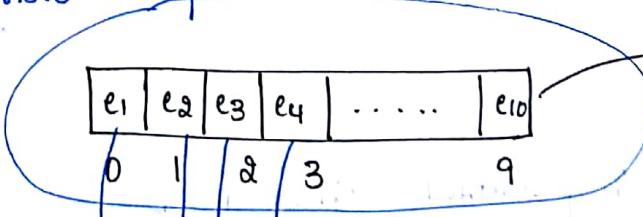
l1.add(e2)

:

l1.add(e10)

l1.add(e11) → grows

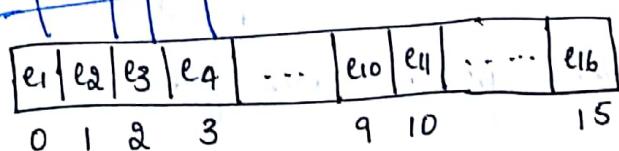
l1.add(e17) → grows



Default capacity

If we try to insert more than it then array list creates object of ArrayList ($10 * 3/2 + 1$)

$$= 15 + 1 \text{ overflow}$$

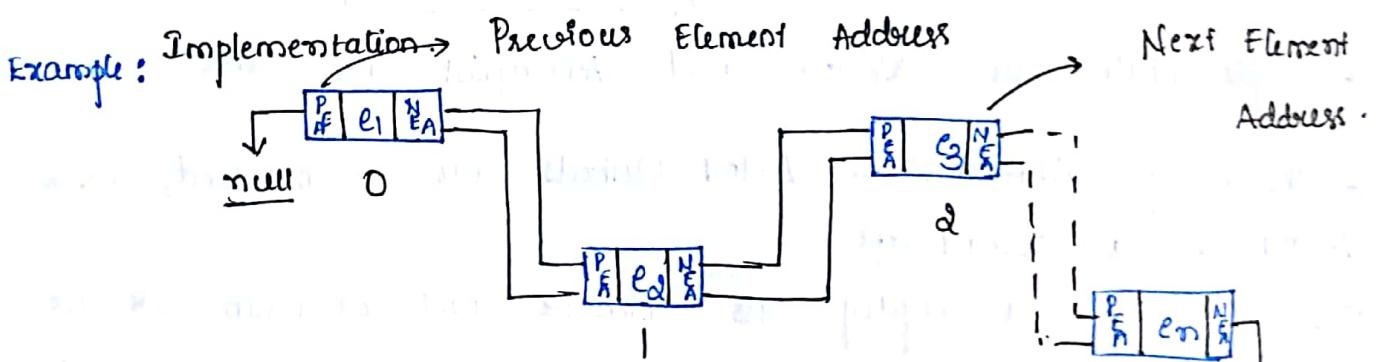


copying takes place

→ Creates the new ArrayList, copies the element present in old ArrayList to the newly created ArrayList and the link li points to newly created ArrayList.

* LINKED LIST

- Linked List is a implementation class of List and Queue Interface.
- It implements two marker Interfaces i.e Serializable & cloneable.
- The linked list is implemented with doubly linked list D.S.
- It grows one node at a time.
- It has only one constructor.
- Linked List is a Non-Sequential List.



→ This does not require copying of elements.

→ ArrayList is preferable whenever we have Retrieval operations more

→ Linked list is preferable when we have Insertion & Deletion more

↳ To retrieve from i^{th} elements, we should ask 3^{rd} element
becoz 3^{rd} element will be knowing the address of i^{th} element.
Hence Retrieval takes much time.

→ Random access of an element in arraylist takes bcz arraylist
implements RandomAccess Interface

→ Random access of an element in linkedlist is not same,
because it does not implement RandomAccess Marker Interface.

* VECTOR

Vector is a implementation class of List Interface.

→ It implements 3 Marker Interfaces If Serializable

& cloneable & RandomAccess

→ The Vector is implemented using resizable Array DS.

→ The default capacity of vector is 10.

→ Vector grows with formula $\lceil \frac{currentCapacity}{2} \rceil$

→ Implementation wise Vector and ArrayList are same.

→ The only difference is Vector Methods are Synchronized, hence
Vector is a Thread Safe.

NOTE: Vector is a legacy class which we won't use much. We use
Vector only when we need a list which is Thread Safe.

** QUEUE

10/10/2017

→ Store and process elements in FIFO.

→ No Index

→ Duplicate is allowed

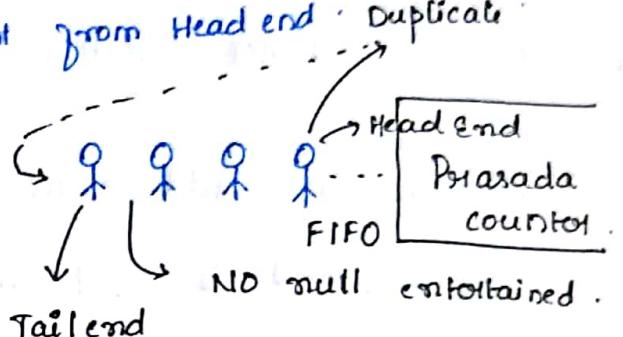
→ Null not allowed



There are two operation:

* Enqueue operation : Adding of element @ tail end

* Dequeue operation : Removing element from Head end Duplicate



** Fundamental Operation of Queue:

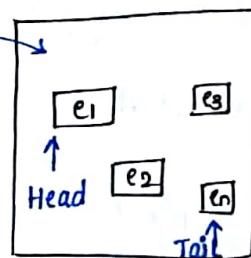
Enque operation

Adding elements

* enqueueing e_1 element

then head & tail pointer will

be pointed to e_1 element.

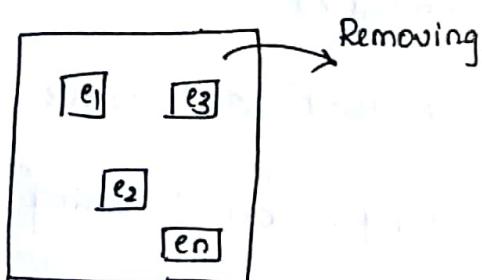


* enqueueing e_2 element then head pointer points to e_1 and tail

pointer points to the e_2 element

* finally the tail pointer points to the last element of enquiring.

Deque operation



** QUEUE INTERFACE

- It is a child Interface of collection Interface, it specifies the operations which can be performed on a Queue.
- The operations are Enque operation & Deque operations.

* QUEUE INTERFACE METHODS:

1) public boolean add (object e)

The method is used to add the specified element to the Queue.

It is used for enqueing operation.

- If Queue is bounded Queue, the method throws IllegalStateException if no space for new element.

2) public boolean offer (object e)

The method is used to add the specified element into the Queue. → It is used for enqueing operation.

- The only diff the add method is it will not throw any Exception if it is a bounded Exception.

3) public Object element ()

The method is used to retrieve head element of the Queue.

- If Queue is empty, on invoking this method on empty queue it throws NoSuchElementException.

4) public Object peek()

The method is used to retrieve head element of the Queue.

→ On invoking this method on an empty queue, the method returns Null.

5) public Object remove()

→ The method is used to remove the head element from the Queue

→ On invoking this method on an empty queue, throws NoSuchElementException.

6) public Object poll()

→ The method removes the head element from the Queue.

→ On an empty queue, the method returns a Null.

* Implementation class of Queue.

→ The Queue Interface has two implementation class.

It Linked List & Priority Queue.

→ Linked List implements Queue Interface Methods and also List Interface Methods.

→ Linked List provides implementations to store the elements with index and also allows to perform Queue based operations.

→ Queue specific operations in the linked list are peek(), boolean,

add(), offer()

7) public boolean addFirst (Object e)

8) public boolean addLast (Object e)

9) public Object getFirst ()

10) public Object getLast ()

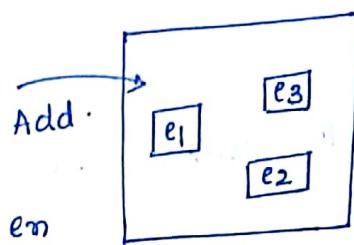
11) public Object removeFirst ()

12) public Object removeLast ()

Concrete Method available only in Linked List.

* Priority Queue class

Enque operation is same but it should satisfy two Rules



Rules: If e1, e2, e3 en

It must be implementation class of Comparable Interfaces

or e1, e2, e3 en. It must be of same type.

Dequeue operation takes place in Ascending Order

1st dequeue operation will remove the least element

and dequeue operation will remove the next least element

→ If duplicates present randomly it remove the element among them

→ In JAVA collection framework, there is no Normal Remove operation

Priority Queue is a class of Queue which satisfies the condition of Comparable

and it is implemented by the class PriorityQueue

Priority Queue is a class of Queue which satisfies the condition of Comparable

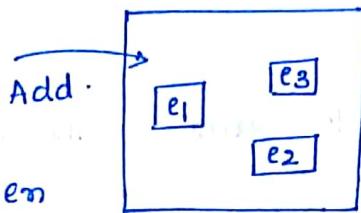
and it is implemented by the class PriorityQueue

Priority Queue is a class of Queue which satisfies the condition of Comparable

and it is implemented by the class PriorityQueue

* Priority Queue class

Enque operation is same but it should satisfy two Rules



Rules: If e1, e2, e3 en

It must be implementation class of Comparable Interfaces

or e1, e2, e3 en. It must be of same type.

Dequeue operation takes place in Ascending Order

1st dequeue operation will remove the least element

and dequeue operation will remove the next least element.

→ If duplicates present randomly it remove the element among them.

→ In JAVA collection framework, there is no Normal Remove operation.

11/10/2017

Example for Queue

```
package queexamples;  
import java.util.PriorityQueue;  
public class PQDemo1  
{  
    public static void main (String [] args)  
    {  
        PriorityQueue q1 = new PriorityQueue ();  
        System.out.println ("Queue Size :" + q1.size ());
```

SOP ("Queue Size :" + q1.size ());

```

SOP ("adding elements to 'q1' ");
q1.add (25); q1.add (12); q1.add (21); q1.add (21); q1.add (5)
q1.add (12);

SOP ( " queue size :" + q1.size ());

SOP (q1);

SOP ( " Dequeuing operation " );
/* Not a good approach
int n = q1.size ();
for (int i=0; i<n; i++)
{
    SOP (q1.poll ());
}
*/
object el = q1.poll ();
while (el != null) {
    SOP (el); el = q1.poll ();
}

```

** Priority Queue is a implementation class of Queue Interface.

→ It implements two Marker Interfaces ~~or~~ Serializable

~~or~~ cloneable.

→ It is implemented with Linked List Queue Datastructure. Where

Dequeueing operation is based on priority. By default the dequeuing is done in a natural Ascending Order.

→ The default capacity of Priority Queue is 11.

→ It grows one node at a time.

→ Constructors of Priority Queue.

1) NO arg Constructor

PriorityQueue q1 = new PriorityQueue()

→ Creates an empty queue with default capacity of 11.

2) Int arg Constructor.

PriorityQueue q1 = new PriorityQueue(n);

n → Specifies initial capacity

→ Creates an empty queue with specified capacity.

3) Collection type arg Constructor.

Let's say c1 is a collection of n elements

PriorityQueue q1 = new PriorityQueue(c1)

→ Creates an PriorityQueue with the elements of the specified collection type. The size of the queue will be the size of the collection.

4) Two arg Constructor

PriorityQueue q1 = new PriorityQueue(c, n);

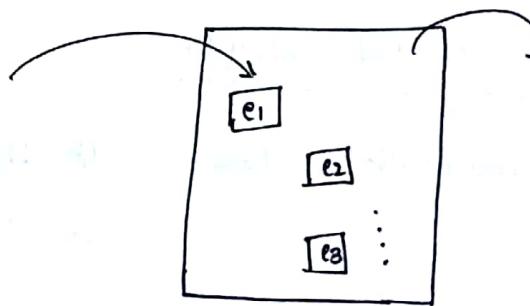
- $C \rightarrow$ Object of Comparator type, specifies Priority Order.
- $n \rightarrow$ int type, specifies initial capacity.
-
- ** SET
-
- \rightarrow A Set is a type of collection which is used to store only Unique elements.
- \rightarrow Set does not allow duplicate elements.
- \rightarrow Elements are stored without index.
- \rightarrow Null Insertion is allowed. (only one Null is allowed since duplicate is not allowed in Set).
- \rightarrow Set elements can be retrieved by using Iterator because there is no specific methods to access the Set.

* Fundamental of Set :

Add operation.

* adding an element e_1

`add (e1);`



retrieve element OR

Remove element use

Iterator .

* adding an element e_2

`add (e2);`

↳ calls

↳ `e2. equals(e1)`

↳ if return value is false, add element e_2 into Set .

↳ if return value is true, add ignore that element

- The Set does not allow duplicate values object on the basis of Hashcode of the object, whenever we add element to the set, add() is implemented to check hashcode of the element to be added with the hashcode of all the elements which are added already into the set. This is done by calling equals() on each object.
- If the equals() method returns true the element will not be added and if it doesn't return true or returns false element will be added to the set.

* SET INTERFACE

- Set Interface is a child Interface of Collection Interface.
- The Set Interface does not specify any specific methods of a Set.

* Implementation class of Set Interface.

- Set has 2 implementation classes
- 1) HashSet
 - 2) TreeSet
- HashSet has a child class or Subclass known as LinkedHashSet.
- HashSet stores unique element but doesn't preserve Order of Insertion.
- LinkedHashSet is a type of HashSet which preserves Order of Insertion.
- TreeSet is a type of Sorted set where the elements are stored in a Sorted Order {Natural Sorting Order (Ascending Order)}.

** Example of Set :

```
package setexamples;
```

```
import java.util.HashSet;
```

```
import java.util.LinkedHashSet;
```

```
import java.util.TreeSet;
```

```
public class Demo1
```

```
{
```

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

```
{
```

```
    HashSet s1 = new HashSet(); // Random
```

```
// LinkedHashSet s1 = new LinkedHashSet(); // Stores the Order
```

```
// TreeSet s1 = new TreeSet();
```

```
    System.out.println ("Set Size : " + s1.size());
```

```
    System.out.println ("adding elements to Set");
```

```
    s1.add(26); s1.add(15); s1.add(85); s1.add(15);
```

```
    s1.add(null);
```

```
    System.out.println ("Set Size : " + s1.size());
```

```
    System.out.println ("Set elements are");
```

```
    System.out.println (s1);
```

```
}
```

→ Set into List or Queue or Any type of collection

by calling a Parameterized constructor

```
// Creating ArrayList with elements of Set s1
```

```
ArrayList l1 = new ArrayList(s1);
```

```
// Creating Queue with elements of Set s1
```

```
PriorityQueue q1 = new PriorityQueue(s1);
```

Null is not allowed in PB

** HASHSET

- HashSet is a implementation class of Set Interface.
- It implements two marker Interfaces i.e Serializable & Cloneable.
- The HashSet is implemented using HashTable [Hash mapped] DS.
- HashSet does not allow duplicate objects based on the hashCode.
- HashSet has a default capacity of 16.
- HashSet grows based on the load factor Or fill ratio.

* Constructors Of HashSet.

1) No arg Constructors

```
HashSet s1 = new HashSet();
```

- Creates an empty HashSet with a capacity of 16 & default load factor is 0.75

2) Two arg Constructors.

```
HashSet s1 = new HashSet(n, f)
```

n → int type , specifies initial capacity .

f → float type , specifies load factor .

- Creates an empty HashSet with the specified initial capacity and load factor .

3) Collection type arg Constructors.

Let's say C1 → Collection with n elements .

HashSet s1 = new HashSet(c1);

→ Guarantees an HashSet with the elements of specified elements of c1 and with a default load factor of 0.75.

* Growing of HashSet and LinkedHashSet:

If we say 'n' → is a capacity of HashSet / LinkedHashSet
(default value of n is 16)

If → is a loadfactor (default value of lf is 0.75)

Grows when the setSize reaches lf * n (lf % of n)
it grows

By default @ 12.

HashSet s1 = new HashSet()

s1.add ("Raju");

s1.add ("Varuu");

:

:

:

12th element s1.add ("Ranju");

UniqueValue

Hashtable / HashMap

	Key	Value
0	# code	Raju
	# code	Varuu
	:	
	:	
15		

The HashSet grows from here.

* LINKED HASHSET:

→ The linked HashSet is a child class of HashSet class.

It is implemented with a hybrid DS which is a combination of HashTable & LinkedList.

→ The HashTable structure is used for unique storage of elements whereas

LinkedList is used to preserve the Order of Insertion.

→ Rest all the properties are same as HashSet.

* TREE SET

→ TreeSet is a implementation class of Sorted Set.

→ The TreeSet implements two Marker Interfaces i.e Serializable & Cloneable.

→ The TreeSet is built on Tree DS.

→ The elements in the TreeSet are arranged in the Natural Sorting Order (Ascending). Q) Does TreeSet allows Null Insertion.

→ TreeSet doesnot allow Null Insertion.

→ If Null is a first element of the TreeSet, then it allows but that TreeSet will not allow another element. It throws NullPointerException.

Exception.

→ We can add only the object which are Comparable type to the TreeSet, otherwise the TreeSet throws ClassCastException.

→ All the elements of the TreeSet must be of same type. Because the TreeSet sorts the elements & stores it in the Tree Structure.

* Constructors of TreeSet

↳ No arg Constructor TreeSet s1 = new TreeSet();

→ Create an empty TreeSet which arranges the element in Natural Sorting order.

→ The TreeSet allows only mutually comparable objects otherwise it throws ClassCastException.

2) Comparator type constructor.

Let's say 'y' is a object of Comparator type

Creates TreeSet s1 = new TreeSet(y);
an empty TreeSet where the elements are not arranged in a Natural Sorting Order but they are arranged as per the implementation of Comparator type.

3) SortedSet type constructor.

Let's say 'x' is a object of type SortedSet.

Creates TreeSet s1 = new TreeSet(x);
a TreeSet with elements of specified SortedSet.

4) Collection type constructor.

Let's say 'c1' is a collection with 'n' elements.

Creates TreeSet s1 = new TreeSet(c1)

Creates a TreeSet with elements of collection type, the elements must be mutually comparable type otherwise Set throws ClassCastException

Example to store user defined Objects.

package listexample;

public class Student

{
int rollno;
String name; double marks;

```
public Student (int rollno, String name, double marks)
{
    this.rollno;
    this.name;
    this.marks;
}

@Override

public String toString()
{
    return "Student [rollno=" + rollno + ", name=" + name + ", marks=" + marks + "]";
}

void addGraceMarks (int grace)
{
    marks = marks + grace;
}
```

NoteBook.java

```
public class NoteBook {

    int pages; double price; String brand;

    public NoteBook (int pages, double price, String brand)
    {
        this.pages = pages;
        this.price = price;
        this.brand = brand;
    }
}
```

```

    @Override
    public String toString()
    {
        return "NoteBook [pages=" + pages + ", price=" + price + ", brand=" + brand + "]";
    }
}

Mobile.java:
public class Mobile
{
    String OS; int ramSize; String brand;

    public Mobile (String OS, int ramSize, String brand)
    {
        this.OS = OS;
        this.ramSize = ramSize;
        this.brand = brand;
    }

    @Override
    public String toString()
    {
        return " Mobile [OS=" + OS + ", ramSize=" + ramSize + ", brand=" + brand + "]";
    }
}

```

//ListDemo1.java

```
import java.util.ArrayList;
import java.util.List;

public class ListDemo1
{
    public static void main (String [] args)
    {
        List li = new ArrayList();
        // Creating a list using ArrayList implementation
        SOP ("adding elements to list");
        li.add (new Student (125, "John", 65.25));
        li.add (new Student (135, "Mike", 75.25));
        li.add (new NoteBook (100, 25.00, "classmate"));
        li.add (new Student (145, "Robert", 56.25));
        li.add (new Mobile ("Android", 4, "LAVA"));

        SOP ("Total elements in list: " + li.size());
        SOP ("List elements are");
        → To remove li.remove (3);

        for (int i=0; i<li.size(); i++)
        {
            SOP (li.get (i));
        }

        SOP ("Adding grace marks all students");
        ExamDepartment examDept = new ExamDepartment ();
    }
}
```

```

> examDept. updateGraceToStudents ( l1, 5 );
> 
> SOP ( " List elements are " );
> 
> for ( int i = 0; i < l1. size (); i ++ )
> {
>     SOP ( l1. get ( i ) );
> }
> }

```

// ExamDepartment.java

```

> import java.util. List;
> 
> public class ExamDepartment
> {
>     void updateGraceToStudents ( List arg1, int arg2 )
>     {
>         for ( int i = 0; i < arg1. size (); i ++ )
>         {
>             Object e = arg1. get ( i );
>             if ( e instanceof Student )
>             {
>                 Student s = ( Student ) e;
>                 s. addGraceMarks ( arg2 );
>             }
>         }
>     }
> }

```

Generalization (Any type of
List AL, LL).

ArrayList arg1

↓

Specialization .

** Example for Storing user defined object in Queue

```
package queueexample;
```

```
public class Student implements Comparable
```

```
{
```

```
    int rollno; String name; double marks;
```

```
public Student (int rollno, String name, double marks)
```

```
{
```

```
    this.rollno = rollno; this.name = name; this.marks = marks;
```

```
}
```

```
@ Override
```

```
public String toString () {
```

```
    return "Student [ rollno = " + rollno + ", name = " + name + ", marks  
        = " + marks + " ]";
```

```
}
```

```
void addGraceMarks (int grace)
```

```
{
```

```
    marks = marks + grace;
```

```
}
```

```
// compare on rollno basis
```

```
public int compareTo (Object arg)
```

```
{
```

```
    Student s = (Student) arg;
```

```
}
```

```
}
```

```
return (int) (this.marks - s.marks); OR  
this.rollno - s.rollno ↪
```

On the basis of Roll
from highest to low by with

(-1)

must be same bcz of
Overriding.

Since marks is
double type casting

```
package queuemethods;

import java.util.PriorityQueue;
import java.util.Queue;

public class PQDemo2
{
    public static void main (String [] args)
    {
        Queue q1 = new PriorityQueue();
        // Queue created using PQ implementation
        System.out.println ("Add elements to Queue");
        q1.add (new Student (145, "John", 65.25));
        q1.add (new Student (130, "Ramu", 14.25));
        q1.add (new Student (120, "Sonu", 70.25));
        q1.add (new Student (125, "Navya", 14.38));
        q1.add (new Student (140, "Pavi", 20.48));

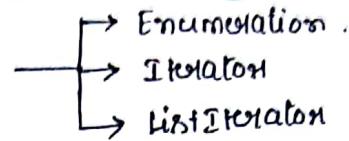
        System.out.println ("Queue size : " + q1.size());
        System.out.println ("Removing elements from the Queue");
        Object o = q1.poll();
        while (o != null)
        {
            Student st = (Student) o;
            System.out.println (st.rollno + "\t" + st.name + "\t" + st.marks);
            o = q1.poll();
        }
    }
}
```

** Cursors In Collection framework

14/10/2014

→ Collection framework library provides Cursors for accessing the elements of any collection in a common way.

→ There are three types of Cursors.



→ The cursors are used to traverse the elements of any type of collection.
↳ moving all one by one.

Cursors

* Enumeration → It is a unidirectional cursor which is used to traverse the elements of any type of collection.

→ It is a legacy cursor.

* Iterator → It is a unidirectional cursor which is used to iterate the elements of any type of collection.

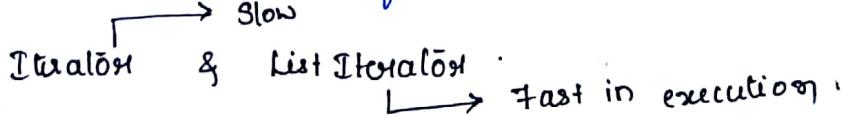
→ This cursor is widely used cursor.

* ListIterator → It is a bidirectional cursor which is used to traverse the elements of List type of collection.

→ It cannot be used on Set and Queue.

→ Using List Iterator we can traverse either in the forward or reverse direction.

→ The methods of the cursor are defined in the Interfaces



** Methods of Iterator Interfaces:

In JCF library,

- Iteration and ListIteration are Interface.
- Library has not exposed the implementation class of these interface used abstraction principle.
- ↳ public Object next()
 - moves the cursor to the next element of collection, returns the reference to that element.
 - If next element is not present throw NoSuchElementException.

↳ public Object remove()

- remove the element where cursor is pointing.

↳ public boolean hasNext()

- return true if next element is present in the collection otherwise returns false.

** Methods of ListIterator Interfaces:

↳ public Object next()

- moves the cursor to the next element of collection, returns the reference to that element.
- If next element is not present throw NoSuchElementException.

↳ public Object remove()

- remove the element where cursor is pointing.

↳ public boolean hasNext()

- return true if next element is present in the collection otherwise returns false.

4) public Object previous()

→ moves the cursor to previous element of collection, returns the reference to that element. If previous element is not present throw `NoSuchElementException`.

5) public boolean hasPrevious()

→ returns true if previous element is present in the collection otherwise returns false.

* HELPER METHODS IN Java Collection Framework.

1) public Iterator iterator() → Collection Interface.

2) public ListIterator listIterator() → List Interface.

* How to use Iterator on Any List?

→ The cursor will be always outside the collection. If we use `next()` the cursor moves to collection.

ArrayList li = new ArrayList();

li.add(e1)

interface

e1	e2	e3	e4	e5
0	1	2	3	4

li.add(e2)

reference

li.add(e3)

Iterator i1 = li.iterator();

li.add(e4)

Object o1 = i1.next();

while (i1.hasNext())

li.add(e5)

Get the

element

Object o2 = i1.next();

{

Object o3 = i1.next();

Object o1 = i1.next();

Object o4 = i1.next();

}

Object o5 = i1.next();

}

when we
want to
check
↓
Next
Element

* How to use Iterator on any list?

e ₁	e ₂	e ₃	e ₄	e ₅
0	1	2	3	4

ArrayList l1 = new ArrayList();

l1.add(e₁);

l1.add(e₂);

l1.add(e₃);

l1.add(e₄);

ListIterator ii = l1.listIterator();

if (ii.hasNext())

{

Object o1 = ii.next();

= Processing code (that make use

of Instance
Keyword)

if (ii.hasPrevious())

{

Object o1 = ii.previous();

=

}

** For-each loop:

→ From JDK 1.5 onwards new loop introduced known as
foreach loop,
→ This loop is used to go through each element of an array or any
collection.

→ The foreach loop internally works on the basis of Iterator.

→ Syntax is for (Object e : referencename)

{

// write code

}

Example: If $l_1 \rightarrow$ is a list of 'n' elements
then

```
for (Object e: l1)
```

```
{
```

// write code

→ here e is referring to the

each element of the list

```
}
```

Collection

Cursor

List

Index

Iterator

List Iterator

for-each loop

Queue

X

✓

X

✓

Set

X

✓

X

✓

Q) Write a java program to store unique records of Students.
Students having duplicate ID should not be allowed. Display
the Student record in tabular column.

// Student.java

```
public class Student
```

```
{
```

```
    int id;
```

```
    String name;
```

```
    double marks;
```

```
    public Student (int id, String name, double marks)
```

add () calls
implicitly hashCode()
() internally.

Uniques required based
on object property then
Overriding of hashCode() &
equals() must.

```

    {
        this.id = id;
        this.name = name;
        this.marks = marks;
    }

    public int hashCode()
    {
        return id;
    }

    public boolean equals(Object arg)
    {
        return this.hashCode() == arg.hashCode();
    }

    public String toString()
    {
        return "Student [id=" + id + ", name = " + name + ", marks=" + marks
               + "]";
    }
}

```

// SetDemo1.java

/* HashSet → choose HashSet
 LinkedHashSet
 TreeSet */

```

public class SetDemo1
{
    public static void main(String [] args)
    {
        HashSet s1 = new HashSet();
        s1.add(new Student(1215, "Ramu", 65.35));
        s1.add(new Student(1213, "Soma", 48.35));
    }
}

```

```
sl.add( new Student (1254, "catu", 95.35);  
sl.add( new Student (1245, "Raju", 49.35);
```

// To avoid duplicate, we need to override hashCode()

```
Iterator i1 = sl.iterator();
```

```
SOP (" . . . - - - - - ");
```

```
SOP (" ID \t name \t marks");
```

```
SOP (" . . . - - - - - ");
```

```
while ( i1.hasNext())
```

```
{
```

```
Student st = (Student) i1.next();
```

```
SOP ( st.id + " \t " + st.name + " \t " + st.marks );
```

```
}
```

```
}
```

15/10/2017

Sunday

** SORTED COLLECTIONS

Queue → Priority Queue (Deque)

↳ Elements are stored in Random Order

* Dequeueing is Ascending Order

Set → TreeSet

* Elements are stored in Sorted Order [Ascending Order]

List → No Sorted List

* Sorting Objects IN Collections:

Sorting of elements takes place as if Compare elements.

Group of elements $e_1, e_2, e_3, \dots, e_m$ → unsorted.

if Arrange elements as Ascending
or Descending

Comparing Techniques: if Using Relational operation.

Arrange in Ascending

$e_1 > e_2$

→ true → e_1 is bigger than e_2

Swap e_1 & e_2 .

→ false → e_1 is Smaller than e_2

No Swapping.

or Using Arithmetic Operation $e_1 - e_2$

result is

→ +ve → e_1 is bigger than e_2 Swap e_1 & e_2 .

→ result is -ve → e_1 is smaller than e_2

No Swap.

→ 0 → e_1 is same as e_2 No swap

→ In Collection framework library Priority Queue & TreeSet

Sorts the elements.

→ Priority Queue Sorts element while dequeuing whereas TreeSet store the element in Sorted Order.

→ By default both are sorting elements in Ascending Order.

- Both the collection allows to store only Mutually Comparable Objects.
(ज्ञानवाला यह compare करने की सुविधा)
- We can add String objects to the Priority Queue or TreeSet because, a String is mutually comparable with another String.
- We can add any wrapper class objects because all wrapper class objects are mutually comparable with another wrapper class object of its type.
- Any class which implements the Comparable Interface are called as Mutually Comparable Objects.
- When a class implements Comparable Interface, a class must provide implementation to the compareTo() on which property the object should be mutually compared.
- When the class gets eligibility to be added into the Sorted collection like Priority Queue and Or TreeSet.

* Example for Sorting PQ using Comparable Interface while Dequeuing.

Employee.java

```
public class Employee implements Comparable  
{  
    int id;  
    String name;  
    double salary;  
  
    public Employee(int id, String name, double salary)
```

```
        this.id = id;
        this.name = name;
        this.salary = salary;
    }
    public int compareTo(Object arg)
    {
        Employee e = (Employee) arg;
        return this.id - e.id;
    }
    public String toString()
    {
        return "Employee [ id = " + id + ..... ] ";
    }
}
```

```
// PQDemo1.java      import java.util.PriorityQueue;
public class PQDemo1
{
    public static void main (String [] args)
    {
        Employee e1 = new Employee (1247, "Sonu", 2500.00);
        Employee e2 = new Employee (1347, "Raju", 4500.00);
        Employee e3 = new Employee (1297, "Mark", 300.00);
        Employee e4 = new Employee (1300, "Patel", 4000.00);
        Employee e5 = new Employee (1427, "Ragini", 2000.00);
```

```
Priority Queue q1 = new Priority Queue()
```

```
q1.add(e1); q1.add(e2); q1.add(e3);
```

```
q1.add(e4); q1.add(e5);
```

```
SOP("-----");
```

```
SOP("ID | Name | Salary");
```

```
SOP("-----");
```

```
Object o = q1.poll();
```

```
while(o != null)
```

```
{
```

```
Employee emp = (Employee) o;
```

```
SOP(emp.id + " | " + emp.name + " | " + emp.salary);
```

```
o = q1.poll();
```

```
}
```

```
}
```

```
}
```

```
-----
```

Comparable : Comparator

→ Mutual Comparison

→ java.lang

Comparator (Non-Mutual)

→ Not Comparison Mutual

→ java.util.Comparator

* Example for using Comparator (Not Mutual Comparison):

```
/* To arrange according to our wish we need to use Comparator
```

```
import java.util.Comparator
```

```
public class BySalaryComparator implements Comparator
```

```
{
```

```
    public int compare(Object o1, Object o2)
```

```
{
```

```
Employee empl = (Employee) o1;
```

```
Employee emp2 = (Employee) o2;
```

```
return (int) (empl.salary - emp2.salary);
```

```
} } // ByNameComparator.java
```

```
import java.util.Comparator;
```

```
public class BySalaryComparator implements Comparator
```

```
{
```

```
public int compare (Object o1, Object o2)
```

```
{
```

```
Employee empl = (Employee) o1;
```

```
Employee emp2 = (Employee) o2;
```

```
return empl.salary - emp2.salary;
```

```
} }
```

→ If we not use this, then we need to write ASCII program to convert each string.

```
// In PQDemo1.class
```

```
: : Employee e5 = new Employee (1451, "Ragini", 2000.00);
```

```
BySalaryComparator salarywise = new BySalaryComparator();
```

```
ByNameComparator namewise = new ByNameComparator();
```

```
PriorityQueue q1 = new PriorityQueue();
```

```
// Mutually arrange in ascending order as per implementation  
of Comparable interface.
```

```
// PriorityQueue q1 = new PriorityQueue(namewise)
```

// Order the elements of queue as per the implementation of
comparator mechanism.

// Priority Queue q1 = new Priority Queue (salarywise)

order the elements of queue as per the implementation of comparison salarywise.

10

4

// In PgDemos using TreeSet.

After Employee e5 =

BySalaryComparator salarywise = new BySalaryComparator();

```
ByNameComparator namewise = new ByNameComparators();
```

```
TreeSet    s1    = new TreeSet();
```

// Mutually arranged in ascending order.

```
// TreeSet s1 = new TreeSet (salarywise);
```

// Arranged as per the implementation of salarywise comparator

```
// TreeSet s1 = new TreeSet (nameList);
```

// Arranged as per the implementation of namewise comparable.

100

Iteration `ii = sl.iterator();` access each element.

```
while (it.hasNext()) {
```

```
Employee emp = (Employee) li.next();
System.out.println(emp.id + " " + emp.name + " " + emp.salary);
}
}
```

* Sorting with List :

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class SortingList
{
    static void displayList(List arr)
    {
        System.out.println(" . . . . . ");
        System.out.println(" ID \tname \tSalary ");
        System.out.println(" . . . . . ");

        for (Object e : arr)
        {
            Employee emp = (Employee)e;
            System.out.println(emp.id + " " + emp.name + " " + emp.salary);
        }
    }
}
```

```

PSVM (String [] args)
{
    ArrayList li = new ArrayList();
    Employee e1 = new Employee (1247, "Ramu", 20000);
    Employee e2 = new Employee (1248, "Soma", 14000);
    Employee e3 = new Employee (1236, "Aishu", 13000);
    Employee e4 = new Employee (1211, "Kempu", 12000);
    li.add (e1); li.add (e2); li.add (e3); li.add (e4); li.add (e5);

    SOP ("Unsorted List");
    displayList (li);

    BySalaryComparator salarywise = new BySalaryComparator ();
    ByNameComparator namewise = new ByNameComparator ();

    Collections.sort (li, namewise);
    SOP ("Sorted List - Name wise");
    displayList (li);

    Collections.sort (li, salarywise);
    SOP ("Sorted List - Salarywise");
    displayList (li);
}
}

```

17/10/2017

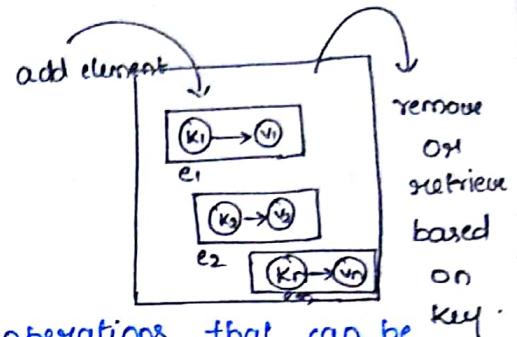
** Map

- Elements are stored in key-value pair.
- Value is mapped to key.

- Key is a index for value.
- Key must be unique. (based on hashCode)
- Value can be duplicate.
- Key can be any type of object.
- Value can be any type of object.

** MAP INTERFACE

- The Map Interface specifies the set of operations that can be performed on a ^{types} map. These operations are declared in a Map Interface. The operations are
- 1) public boolean put (Object key, Object value)
 - The method is used to add the specified pair of object into the Map.
 - The first argument is key, the second argument is value.
 - The value is mapped to the key and stored in the Map.
 - If the key is already present in the Map, it replaces the value mapped to the key.
- 2) public Object get (Object key)
 - The method returns the value which is mapped to the specified key.
 - If key is not found, returns null.
- 3) public Object remove (Object key)
 - The method removes the entire key value pair from the map based on the key specified.
- 4) public boolean containsKey (Object key)
 - The method returns true if the key specified is found in the map.



Otherwise it returns false.

5) public boolean containsValue(Object value)

The method returns true if the specified value is found in the Map. otherwise it returns false.

6) public Set keySet()

The method returns set of all the keys present in the Map, in a set format.

7) public Collection values()

The method returns collection of value from the Map in a collection format.

*IMPLEMENTATION CLASS:

The Map Interface has 3 implementation class.

1) HashMap 2) HashTable 3) TreeMap.

→ HashMap stores the element in key-value pair. It allows Null as a key and Null as a value.

* It doesnot preserve the order of Insertion.

→ The LinkedHashMap is a subclass of HashMap which preserves the order of insertion.

→ HashTable allows key-value pair but key and value should not be Null.

* It is a threadsafe class.

* HashTable is also called as a Synchronized Map.

→ TreeMap is a type of Map which implements Sorted Map according to key.

Hence the elements in TreeMap are sorted in Natural Ascending Order.

```

EXAMPLE : -----
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Hashtable;

public class MapDemo1
{
    public static void main (String [] args)
    {
        HashMap m1 = new HashMap();
        m1.put ("java", "java");
        m1.put ("pi", 3.142);
        m1.put ('e', true);
        m1.put (false, 12876);
        m1.put (null, 154);

        SOP( "Map Size: " + m1.size());
        SOP( "Map Elements :");
        SOP( m1);
        → SOP( "value mapped to 'pi' : " + m1.get("pi"));

        Set s1 = m1.keySet(); → To print all the
        Iterator i1 = s1.iterator();
        while (i1.hasNext())
        {
            Object key = i1.next();
            Object val = m1.get(key);
            SOP( key + "---->" + val);
        }
    }
}

```

O/P
 - - - - . . .
 B → 12
 D → 10
 T → 22
 X → 2 . . .

To Map each
 element in order

** CLONING :

```
class Demo1
{
    int x = 12;
    double y = 4.5;
    void display()
    {
        SOP(x);
        SOP(y);
    }
}
```

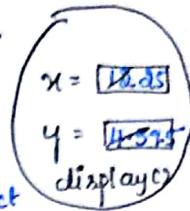
`New Operator` → creates an instance of a class.

`Demo1 d1 = new Demo1();` →

`d1.x = 25;`

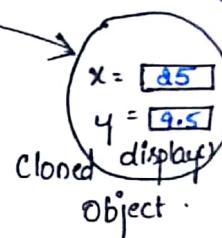
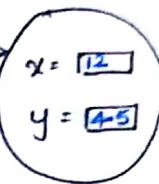
`d1.y = 9.5;`

Original object



`Demo2 d2 = new Demo2();` →

`d1.clone();`



→ `clone()` is used to make only copy of already existed

~ Object not the class.

* protected Object clone()

→ Method defined in `Object` class.

→ It is implemented to take a clone of the object on which the method is invoked.

→ This method uses shallow copy cloning.

→ We can take clone of an object, if it is a type of "cloneable".

Example: `Mobile.java` // SHALLOW COPYING.

`package pack1;`

```
public class Mobile implements Cloneable
{
    int name; String os; int ramSize;
```

```

    public Mobile (int imie, String os, int ramSize)
    {
        this.imie = imie; this.os = os; this.ramSize = ramSize;
    }

    public String toString()
    {
        return "Mobile [imie=" + imie + ", OS=" + os + ", ramSize=" + ramSize];
    }

    public Object clone () throws CloneNotSupportedException
    {
        return super.clone();
    }
}

// Mainclass.java.

package pack1;

public class Mainclass
{
    psvm (String [] args)
    {
        Mobile origMobile = new Mobile (32154, "Android", 4),
        Mobile cloneMobile = null;

        try {
            cloneMobile = (Mobile) origMobile.clone(); // return type is object. Hence we need to refer
            // to access the properties.
        }
        catch (CloneNotSupportedException e)
        {
            SOP ("clone cannot take a clone of the object");
        }
    }
}

```

QUESTION

```

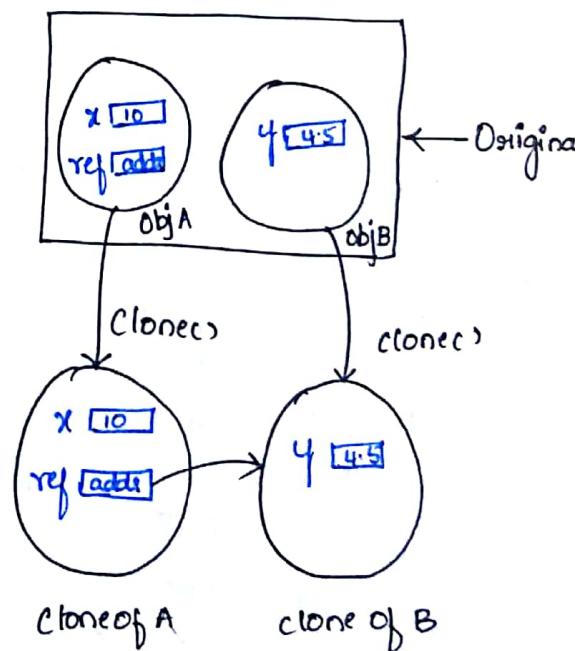
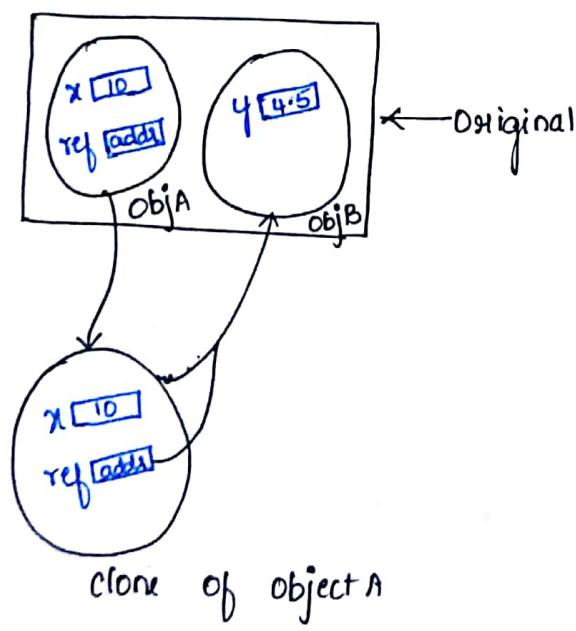
SOP ("hashcode number of original mobile: " + origMobile);
SOP ("hashcode number of cloned mobile: " + cloneMobile);
SOP ("Original mobile properties");
SOP (origMobile);
SOP ("cloned mobile properties");
SOP (cloneMobile);
}

```

** TYPES OF CLONING:

* SHALLOW COPY

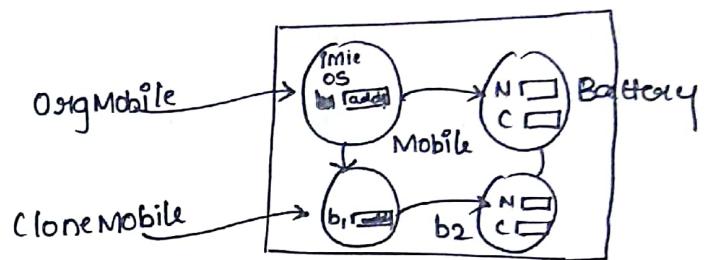
* DEEP COPY.



- In an object class the code is written only for Shallow copy.
- All collections are Deep copying.
- For Deep copy we need to override public Object clone().

Example for DEEP COPYING.

```
public class Mobile implements Cloneable  
{  
    .  
    :  
    public Object clone() throws CloneNotSupportedException  
    {  
        Mobile m1 = new Mobile(this.imie, this.os, this.ramSize);  
        Battery ba = new Battery(this.bl.number, this.bl.capacity);  
        m1.bl = ba;  
        return m1;  
    }  
}
```



//Battery.java.

```
package pack1;  
  
public class Battery  
{  
    int number; int capacity;  
  
    public Battery (int number, int capacity)  
    {  
        this.number = number;  
        this.capacity = capacity;  
    }  
  
    public String toString()  
    {  
        return "Battery [number = "+ number + .....]";  
    }  
}
```

// Mainclass.java

```
public class Mainclass  
{  
    :  
    :  
}
```

```
SOP (" hashCode number of battery :" + origMobile.bl.hashCode()); } Diff.  
SOP (" hashCode number of battery : " + cloneMobile.bl.hashCode()); } Hashcode  
SOP (origMobile.bl);  
SOP (cloneMobile.bl);  
  
SOP (" Original Mobile properties");  
SOP (origMobile);  
SOP (" cloned Mobile properties");  
SOP (cloneMobile);  
}  
-----  
19/10/2017
```

Example: Creating a Rectangle from Insert and copying the created Rectangle.

→ If we do not want to disturb the original then cloning is required.

→ To avoid repetition of contacting Server. we make use of cloning.

→ If we want to make changes in the client side. we can create a copy of the objects perform actions. If succeed then apply to original else destroy them or create new clone.

For ex There are 12 months in a year

but we can declare or initialize the variable to 13 also. Hence these are unsafe So, we are going to make use of Enum type.

Variables

```
int x ;  
int y ; } Primitive  
char z ;  
String s1 ; → Non-Primitive
```

type VariableName
Data type it class type Ex: int x=13; int x=20;
 or Interface type
 or Enum type

ENUM TYPE Example:

```
package pack1;  
public enum Day  
{  
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;  
}  
  
// Main class.java  
package pack1;  
public class Mainclass  
{  
    public static void main (String [] args)  
    {  
        System.out.println ("main method started");  
  
        Day dayName; → classname. MemberName .  
        dayName = Day.THURSDAY;  
  
        System.out.println ("Today is :" + dayName);  
        System.out.println ("main method ended");  
    }  
}
```

Industry Standard writing in caps



Static & Final Members

Interface Type



Specify the behavior
(what)

Class Type



define the behavior
object (How)

Enum Type



Define set of Constant
in an appm.

When and Where → Polymorphism.

- Enum does not inherit from Object class
- Does not have constructors bcos it is not used for obj creation

** GENERICS

→ Does not support primitive type.

Normal code

```
public class Student  
{  
    int id;  
  
    public Student (int id)  
    {  
        this.id = id;  
    }  
  
    public int getId()  
    {  
        return id;  
    }  
  
    public void setId (int id)  
    {  
        this.id = id;  
    }  
}
```

Generics code

```
public class Student <T>  
{  
    T id;  
  
    public Student (T id)  
    {  
        this.id = id;  
    }  
  
    public T getId() {  
        return id;  
    }  
  
    public void setId (T id)  
    {  
        this.id = id;  
    }  
}  
// Main class.
```

→ changing the datatype @ runtime is known as Generics.

```
-----  
Student <String> s2 = new Student  
<String> ("a764b7");  
s2.setId ("REVISEDNOT");  
SOP ("Student ID:" + s2.getId());  
}
```

```
public class Mainclass  
{  
    PSVM (String [] args)  
    {  
        Student <Integer> s1 = new Student  
<Integer> (3215);  
        s1.setId (321546);  
        SOP ("Student ID:" + s1.getId());  
    }  
}
```

* Generics in Collection:

```
public class MainClass  
{  
    public static void main (String [] args)  
    {  
        ArrayList li = new ArrayList();  
        // type unsafety  
        li.add (32486);  
        li.add ("Raju");  
    }  
}
```

// Type Safety in collection does not convert elements into object, it remain as it is.

secA should contain Student type of string values.

```
ArrayList < Student < String >> secA = new ArrayList < Student < String >>;
```

```
secA.add (new Student < String > ("CS1248"));
```

} (secA.add ("CS124") ; // throws error because this is safety type.

} secA.add (new Student < Integer > (3424)); // throws error bcz

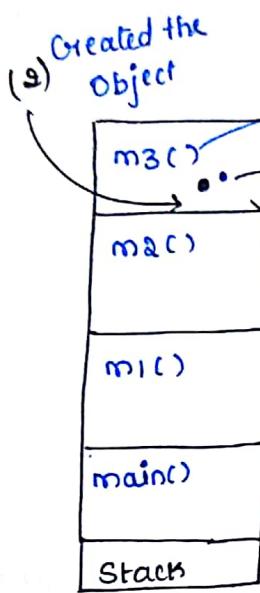
ArrayList is of Student with String type.

** EXCEPTIONS IN JAVA

23/10/2017

Function / Method Execution: Method execution happens on Stack.

Ex: main() → m1() → m2() → m3()



Call Stack
(LIFO)

- (1) Event, disturb the method execution.
- (2) Handover the object to Runtime System.
(looks for the block of code which can handle the object)
- (3) Looks for block of code in the current method, if found execute the code.

→ In JVM, the method execution happens on the stack.

- * Any called method enters on the top of the stack and performs its operation, exit from the stack.
- * The control of execution will be always with the method which is on the top of the stack.
- * The stack follows the Last In First Out (LIFO).

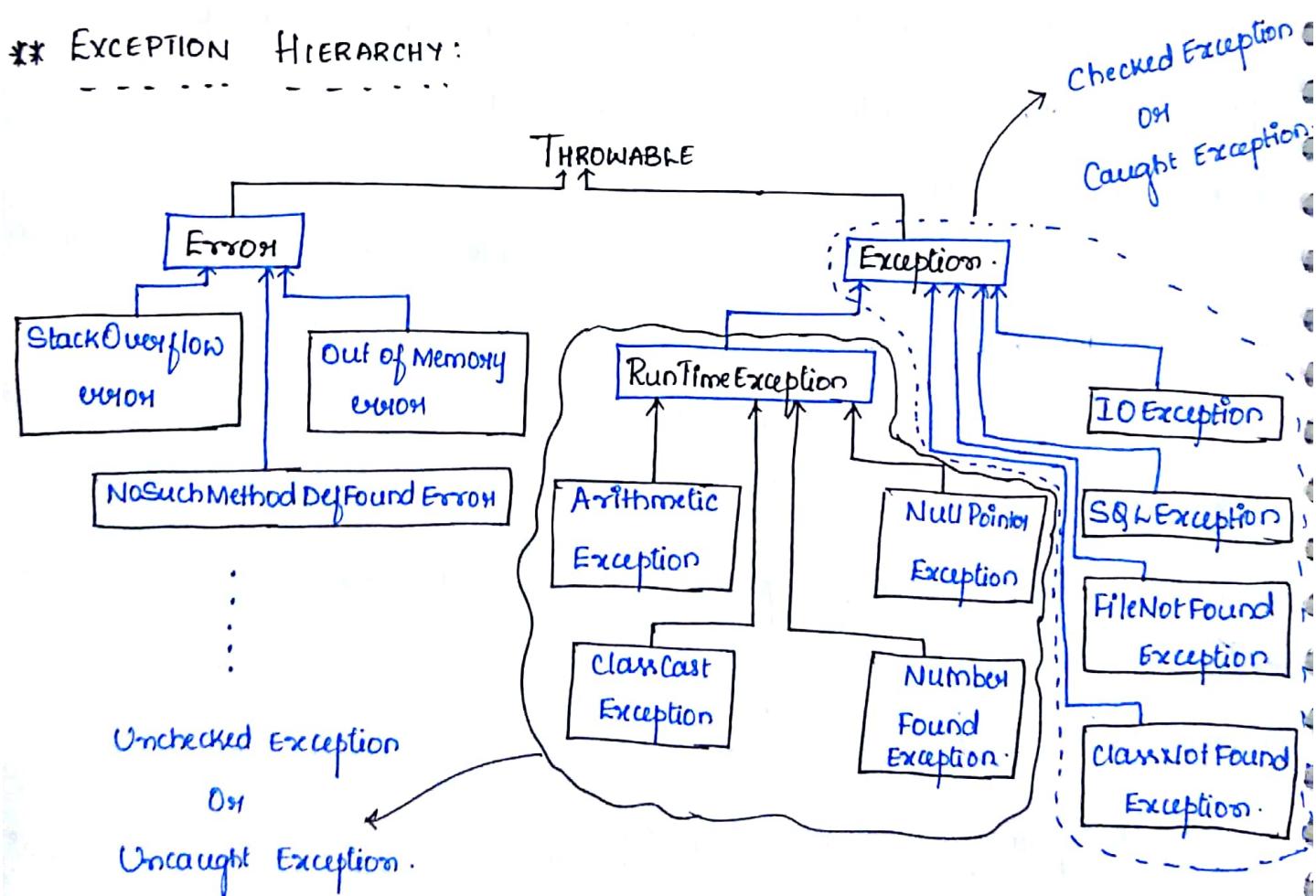
→ When a method is running an event might disturb an executing because of this event, the method creates an object, and hands over to the runtime system.

- * The object is known as EXCEPTION and the process is known as RISE OF EXCEPTION.
 - Once the Runtime receives an object from the method, the runtime system looks for the block of code which can handle the object or situation.
 - The block of code which runs when an exception occurs is known as EXCEPTION HANDLER.
 - The runtime system always look for the Handler in the Current Method, if found runs the Handler code, if not found, looks for the handler, in the caller Method.
- * In the Caller Method also, if Handler is not defined, the runtime system looks in its caller Method, likewise, it looks in the entire caller stack for the Handler.
- * If Handler is not found, then runtime system terminates the execution by shutting down the JVM, this might cause loss of data.
- During compilation of Java code, the compiler can anticipate the exception or event based on the operation.
- * If the compiler successfully anticipates, then it forces the programmer to define the Handler in the current Method, otherwise, compiler throws an error, "Unhandled Exception error".
- * The exception which is anticipated by the compiler during compilation time is known as CHECKED EXCEPTION OR CAUGHT EXCEPTION.
- * The exception which are not anticipated by the compiler

→ Example : If we define a method to perform division operation by reading a values from the user, the compiler compiles the division expression without checking values, but while running program if the denominator value is 0, then JVM throws an exception known as Arithmetic Exception.

→ If we define a method to read a data from a file, then a compiler anticipate, if the file is not found in the specified path what is the alternative and compiler forces the programmer to define handler for FileNotFoundException.

** EXCEPTION HIERARCHY :



** How to write Exception Handling Programs?

24/10/2017.

```
try  
{  
    = Operation  
}  
  
}  
Exception classname which need to be handled.  
Catch (argument arg) → No statements should be written in between  
{  
    = } Block of code OR Handler.  
try/catch block.
```

Example: calculator.java.

```
package pack1;  
public class calculator  
{  
    void divide (int n1, int n2)  
    {  
        SOP ("dividing "+n1+" by "+n2);  
        int res = 0; // local variables must  
                    be initialized.  
        try {  
            SOP ("try block started");  
            res = n1/n2;  
            SOP ("try block ended");  
        }  
        catch (ArithmaticException exp)  
        { SOP ("cannot divide "+n1+" by "+n2+ " hence dividing by 0"); res = n1; } }
```

Mainclass.java

```
public class Mainclass  
{  
    calculator c1 = new calculator();  
    c1.divide (12, 2);  
}
```

↓ Try block is executed

If c1.divide (12, 0);

↓ Catch block is executed

```
{ }
```

* Handling the Exception in Caller Method.

```

public class Mainclass
{
    public void (String [] args)
    {
        Calculator c1 = new Calculator ();
        try { c1.divide (12,2); }
        catch { (ArithmeticalException exp) { SOP ("cannot divide by zero"); } }
    }
}

public class Calculator
{
    void divide (int n1, int n2)
    {
        int res = 0;
        res = n1/n2;
        } } SOP ("result is" + res);
}

```

O/P : It will be executed.

* Try with Multiple catch Blocks:

```

public class Calculator
{
    int divide (int n1, int n2)
    {
        SOP ("dividing " + n1 + " by " + n2);
        int res = 0;
        res = n1/n2;
        return res;
    }
}

```

Defining an array.

Storing the result in
4th position.

If we try to insert in
5th position, and catch
block is executed.

```

Mainclass.java
public class Mainclass
{
    public void (String [] args)
    {
        int [] arr1 = new int [5];
        Calculator c1 = new calculator();
        try {
            arr1[3] = c1.divide(12,2);
        }
        catch (ArithmeticalException exp)
        {
            SOP ("cannot divide by 0");
        }
        catch (ArrayIndexOutOfBoundsException exp)
        {
            SOP ("index out of boundary");
        }
        SOP ("element is" + arr1[3]);
    }
}

```

* Try, catch and finally blocks:

```
try {  
    SOP ("try block started");  
    arr[6] = c1.divide (1a, a);  
    SOP ("try block ended");  
}  
  
catch (ArithmaticException exp) { SOP ("cannot divide by 0"); }  
  
catch (ArrayIndexOutOfBoundsException exp) { SOP ("Index out of boundary") }  
  
finally { SOP ("I always run"); }  
SOP ("element is " + arr[3]);
```

try { }
catch (...) { }
finally { }
DB
FileHandling
Server

→ We cannot write only try block. Try block must be followed by catch block OR finally block.

→ try { } → Once we write finally block, it does not handle the exception. but it executes some set of code before shutdown.
finally { } → If we donot know the exception, before JVM shutdown, it must execute the code in finally block.

```
public class Sequence  
{  
    void printNumbers (int start, int end)  
    {  
        SOP ("Printing numbers from " + start + " to " + end);  
        for (int i = start; i <= end; i++) { SOP (i);  
            * Thread.sleep (1000); }  
    }  
}
```

Compiler shows red mark. ie compiler is anticipating to handle write try & catch block ie checked exception.

* Example showing Compiler Anticipation:

```
public class Sequence  
{  
    void printNumbers (int start, int end)  
    {  
        for (int i = start; i <= end; i++)  
        {  
            SOP (i);  
        }  
        try {  
            Thread.sleep(1000);  
        }  
        catch (InterruptedException e)  
        {  
            SOP ("interrupted");  
        }  
    }  
}
```

* Main class

```
public class Mainclass  
{  
    public static void main (String [] args)  
    {  
        Sequence s1 = new Sequence();  
        s1.printNumbers (1, 25);  
    }  
}
```

25/10/2017

↳ try { or try {

 }
}

catch (ArithmeticExp)

```
{  
    }  
}
```

↳ Handle only ArithmeticExp
 " Specific Handler "

catch (RuntimeExp)

```
{  
    }  
}
```

→ can handle all RuntimeException
 " Generic Handler "

```

try {
    ...
}
catch (Exception ex) {
    ...
}

```

can handle any Exception

"Generic Handler"

```

try {
    ...
}
catch (Throwable th) {
    ...
}

```

can handle both Error and Exception

"Generic Handler".

* Hierarchy of Catch blocks:

```

try {
    ...
}
catch (ArithmaticException exp) {
    ...
}
catch (ArrayIndexOutOfBoundsException ex) {
    ...
}
catch (Exception ex) {
    ...
}

```

Always checks 1st, 2nd... last catch block.

Specific Handler
must be always @ last.

```

try {
    ...
}
catch (Exception ex) {
    ...
}
catch (Exception ex) {
    ...
}
catch (Exception ex) {
    ...
}

```

This can handle all types of exception
 \therefore rest becomes waste. Hence we need to write Generic Handler @ last.

** throw statement:

→ Used to throw an object of Throwable type.

throw means Handover the exception to Runtime System.

Syntax: throw e;

e → an object of throwable type.

Ex: (1) throw new ArithmeticException()

(2) ArithmeticException e1 = newArithmeticException();
throw e1;

(3) throw new String() → Error because String is not a
type of throwable (only errors
and Exceptions).

** Rules: It can throw only throwable type object.

It only one object can be thrown.

It must be used in Method body or Constructor body.

** Own methods to throw errors / Exception.

```
public class calculator
```

```
{
```

```
void divide (int n1, int n2)
```

```
{
```

```
if (n2 == 0)
```

```
{
```

```
throw new ArithmeticException ("Denominator can't be 0")
```

```
}
```

Two types Arg:
1) String arg type con
2) No arg con.

```

        else
    {
        SOP ("dividing " + m1 + " by " + m2);
        int res=0;
        res = m1/m2;
        SOP ("result is " + res);
    }
}

// Main class.

public static void main (String [] args)
{
    SOP ("main method started");
    calculator c1 = new calculator();
    try {
        c1.divide (10,0);
    }
    catch (ArithmeticException exp)
    {
        SOP ("printing exception details");
        exp.printStackTrace();
    }
    SOP ("main method ended");
}

```

If we throw checked Exception then it looks
 ** handler in Current Method
 Checked exception always look
 for handler in Caller Method
 It looks only in calling Method

* Unchecked exception looks
 in caller Method

**
 exception are always thrown
 @ Runtime.

**** throws Keyword:**

```
public class calculator
{
    void divide (int m1, int m2) throws SQLException;
    {
        try if (m2 == 0)
        {
            throw new SQLException ("Denominator can't be 0");
        }
        else {
            SOP ("dividing " + m1 + " by " + m2);
            int res = 0;
            res = m1 / m2;
            SOP (res);
        }
    }
}

import java.sqlException;

public class Mainclass
{
    psvm (String [] args)
    {
        SOP ("main method started");
        calculator c1 = new calculator ();
        try {
            c1.divide (12, 0);
        }
        catch (SQLException exp)
```

for checked Exception to look
Handle in caller Method

```

    {
        SOP("printing exception details");
        exp.printStackTrace();
    }
    SOP("main method ended");
}
}
-----

```

- throw statement can throw any type of throwable type object.
- Whenever an unchecked exception category object is thrown, the runtime system looks for the handler in the current method, if not looks for the handler in the caller method.
- Whenever the throw statement throws checked exception category object, the runtime system looks for the handler in current body, if not compiler throws error.
- We can specify the caller to define the handler by using throws declaration keyword.
- The throws declaration keyword can specify more than one exception it must be used only in the method declaration as well as Constructors.
- The throws keyword must be used for Checked Exception.

* JAVA has a default Handler.

86/10/2017

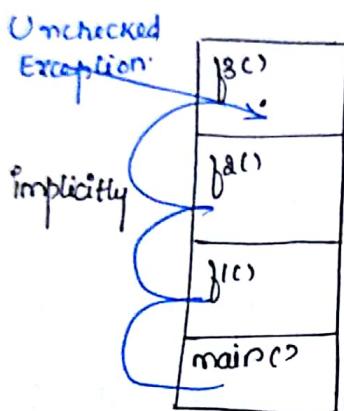
* If we won't specify the Handler then Runtime System calls Default Handler.

- throws never create the object.
- If no one handling, JVM calls Default Handler.

* Throws Declaration

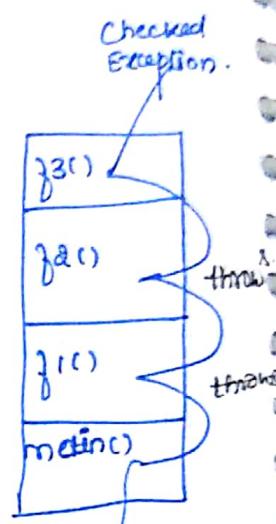
Ex: Method throwing Unchecked Exception object

main() → f1() → f2() → f3()



R-T System calls "Default Handler"

↓
Stop, display stack addresses.



Stack Unwinding: It collapses all the triple stack before shutting down.

Example:

```
public class DemoA
{
    void f2() throws FileNotFoundException
    {
        throw new FileNotFoundException ("File is not found in the path.");
    }
}

public class DemoB
{
    void f1() throws FileNotFoundException
    {
        DemoA d1 = new DemoA();
        d1.f2();
    }
}
```

```

import java.io.FileNotFoundException;
public class Mainclass
{
    public sum (String [] args) throws FileNotFoundException
    {
        Demo d1 = new Demo1();
        d1.f1();
    }
}

```

throw statement throws
only one object @ a time.
throws can specify
multiexception { more than
one exception.}

If we want to use throw statement , following code explains

```

import java.io.FileNotFoundException;
import java.sql.SQLException;
public class Demo2
{
    void f1() throws FileNotFoundException, SQLException
    {
        int x = 12;
        if (x > 10)
        {
            throw new FileNotFoundException ("File is not found in the path");
        } else
        {
            throw new SQLException();
        }
    }
}

```

∴ In f1()
also we need
to implement
the same declaration ,

** CUSTOM EXCEPTION OR USER DEFINED EXCEPTION:

at 10/10/2017

```
class XYZ
{
    members
}

class PQR extends RuntimeException
{
    members
}
```

throw new XYZ(); *exception, not type of Throwables.*

throw new PQR(); *→ works,*
bcoz, PQR is a type of Throwable

* User Defined Unchecked Exception:

InsufficientBalanceException = IBE

```
public class InsufficientBalanceException extends RuntimeException
{
    private String message;

    // no arg constructor.

    public IBE()
    {
        message = " ";
    }

    // String arg constructor

    public IBE (String message)
    {
        this.message = message;
    }
}
```

```
    public String getMessage()
    {
        return
    }
```

```

'; Account.java
;
public class Account
{
    String name;
    double accbal;
    public Account (String name, double accbal)
    {
        this.name = name;
        this.accbal = accbal;
    }
    deposit
    public void (double amt)
    {
        SOP ("Depositing Rs" + amt);
        accbal = accbal + amt;
    }
    public void withdraw( double amt)
    {
        SOP ("Withdrawing Rs " + amt);
        // if balance is not sufficient. throw exception
        if (amt > accbal)
        {
            throws new InsufficientBalanceException ("Current Balance :" + this.accbal);
        } else {
            accbal = accbal - amt;
        }
    }
    public void checkBalance() { SOP ("Account Balance :" + accbal);
}
}

```

Mainclass

```
public class Mainclass
{
    public sum (String [] args)
    {
        SOP ("main method started");

        Account a1 = new Account ("Raju", 10000.00);
        a1 . checkBalance ();
        a1 . deposit (5000.00);
        a1 . checkBalance ();

        try {
            a1 . withdraw (6000.00);
        }
        catch (IBE e)
        {
            SOP ("unable to withdraw");
            SOP (e.getMessage ());
        }
        a1 . checkBalance ();
        SOP ("main method ended");
    }
}
```

** User-Defined Checked Exception

```
public class InsufficientBalanceException extends Exception
{
}
```

```

public void withdraw (double amt) throws InsufficientBalanceException {
    SOP (" withdrawing Rs " + amt );
    if (amt > accbal)
    {
        throw new InsufficientBalanceException ("Current Balance: " + this.acc);
    }
    else {
        accbal = accbal - amt;
    }
}

```

* Single task Application:

→ at any point of time only one task

Ex: calculator, command prompt.

* Multi task Application:

→ at any point of time more than one task. Ex: Gmail, Music player

1) Process Mode → Heavy weight, consumes more memory. Slow in execution

2) Thread Mode → Light weight, consumes less memory, fast in execution.

JVM → Runs program in Thread Mode.

OR

parallel Execution

Games are all
Multithreading OR Thread Concurrency

** THREADS

- Thread class defines the properties of Thread.
- Thread class is defined in java.lang package.
- The Thread class specifies 3 properties 1) ID 2) Name 3) Priority.
- The Thread class has overloaded constructors.
- The Thread class defines set of behaviour which can be operated on the properties.

** THREAD PROPERTIES.

- Each and every thread created in the JVM must have the foll properties.
- * THREAD ID → It is a unique Integer number, which is used to identify thread class instance created in JVM.
- * The Thread Id is initialized at the time of Object creation automatically.
- We cannot change the Id of a thread.

THREAD NAME : → It is a string type used to set a name of a thread.

For every thread created in the JVM, default name is initialized.

User can provide name to the thread and ^{can} change the name of the thread.

THREAD PRIORITY → The thread priority is a number which is used to define priority of a thread.

- The priority ranges from 1 (low) to 10 (High).
- If user is not defining a priority then default priority is 5 (mid).
- The JVM executes the thread based on the thread priority.

**Constructors of Thread class

→ The Thread class has Overloaded constructors. The few important constructors are:

* NO arg Constructor

Thread t1 = new Thread();

object

A thread class is created with the default values of thread properties.

* String arg type Constructors

Thread t1 = new Thread("add");

Creates a thread class object where thread name will be add
thread id and thread priority gets a default initialization.

* Runnable arg type Constructors

Let's say r1 is a object of Runnable type then

Thread t1 = new Thread(r1);

Creates a thread class object with runnable type object.

* 2 arg Constructors

1st arg → Runnable type and arg → String type

Thread t1 = new Thread(r1, "Add");

Creates thread class object with the specified runnable type &
specified thread name.

** METHODS OF THREADS :

1) public int getId()

On invoking this method, it returns the ID of the ^{current} thread.

2) public String getName()

On invoking this method, it returns name of the thread.

3) public int getPriority()

On invoking this method, it returns the priority number of the thread.

4) public void setName(String name)

The method is used to set the name of the thread.

5) public void setPriority (int priorityNum)

The method is used to set the priority number of a thread.

6) public void start()

On invoking this method, it starts execution of thread. The start() internally calls the run method of the thread object.

7) public void run()

This method is used to define the thread task. Whenever we start a thread, the thread execution begins by calling Run().

8) public void stop()

The method is used to stop the running thread. On invoking this method, the running thread will be terminated.

9) public static Thread currentThread()

On invoking this method, it returns reference to the current running thread. We can use this () in any context.

static

10) public void sleep (long time)

On invoking this method, it pauses the current running thread for the specified period of time. Once the time is elapsed, the thread resumes the execution.

```
package pack1;

public class Mainclass
{
    public void main (String [] args)
    {
        Thread t1 = new Thread () ; // creating a thread .
        SOP ("thread id:" + t1.getId ());
        SOP ("thread Name:" + t1.getName ());
        SOP ("thread priority :" + t1.getPriority ());
        SOP ("setting thread name");
        t1.setName ("CAD- Thread");
        SOP ("setting thread priority");
        t1.setPriority (7);
        SOP (" thread name :" + t1.getName ());
        SOP ("thread priority :" + t1.getPriority ());
    }
}
```

Example2 : package pack1 :

```
public class Sequence1 extends Thread
{
    public void run()
    {
        SOP ("printing numbers from 1 to 25 ");
        for (int i=1; i<=25; i++)
        {
            SOP (i);
        }
    }
}
```

```

SOP ("i=" + i);

try {
    Thread.sleep(1000);
}
catch (InterruptedException e) {
    e.printStackTrace();
}
}
}
}

```

O/P:

```

1
101
102
2
3
103
:

```

→ When the operations are different we need to create new class of Thread.

Multi-thread performing different task.

public class Sequence2 extends Thread

```

{
    public void run()
    {
        SOP ("printing nos from 101 to 125");
        for (int j = 101; j <= 125; j++)
        {
            SOP ("j=" + j);
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}

```

Mainclass

```

public class mainclass
{
    public static void main (String [] args)
    {
        SOP ("main method started");

        Sequential s1 = new Sequential();
        //Creating thread
        Sequential s2 = new Sequential();
        //Creating thread.
        s1.start();
        s2.start();
        SOP ("main method ended");
    }
}

```

→ On invoking start() on a thread, the thread is registered to the thread scheduler of jvm, Once Jvm allocates new stack, it calls run() to do the execution on the New Stack.

→ If we directly invoke the run(), instead of start() then the execution happens on the current Stack which will not allow achieve parallel execution.

30/10/2014

** RUNNABLE INTERFACE:

→ defined in java.lang

* Only one method.

public void run()

→ A class implementing runnable interface is known as "Runnable Type Object".

Example: Sequence3.java //ways of creating thread using Runnable Interface is a relations

package pack1;

public void run()

{

SOP ("printing numbers from 201 to 225");

for (int k=201; k<=225; k++) {

{

SOP ("k=" + k);

try {

Thread.sleep (1000);

}

catch (Exception e) {

```

public class MainClass
{
    public void main (String [] args)
    {
        System.out.println ("main method started");

        Sequence3 s1 = new Sequence3(); // Runnable type
                                         → s1.start() → throws error.
        Thread t1 = new Thread (s1);
                                         ↓ Runnable type object.

        Sequence1 s2 = new Sequence1(); // Is-A relationship

        t1.start(); // start will call run() of Runnable type object
        t2.start(); // Thread type object.

        System.out.println ("main method ended");
    }
}

```

** // Sequence3 is type of Thread & Runnable.

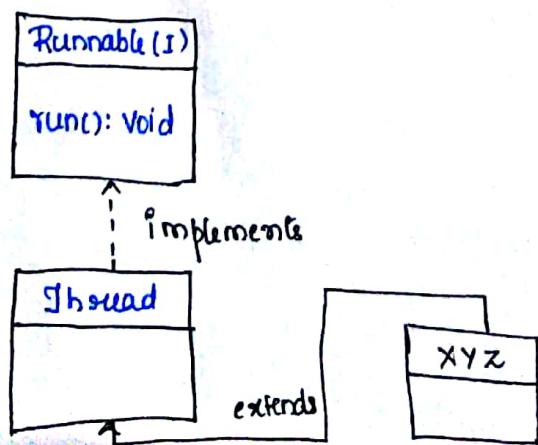
public class Sequence3 extends Thread
 implements Runnable.

First
extends then
implements.

NOTE:

Thread class is also an implementation class of Runnable interface which provides an empty implementation of run().

→ Thread class is a implementation class of Runnable Interfaces.



XYZ has both properties of Run & Thread.

→ threads created inside JVM automatically.

* Whenever we start JVM execution, it creates 3 threads internally.

1) Thread main

2) Thread Scheduler.

3) Garbage Collector Thread.

→ The Thread main starts the program execution by calling main method of a class given to the JVM.

* Thread main always makes a static reference to the main() of the given class. Hence the main method must be always static.

* The Thread main reads the command line arguments, stores it in the String array and passes that array to the main method. Hence the main method argument must be String type array.

Syntax: `java classname [values]`

↓ ↓ ↓ ↓ ↓
Command class to be values values values
name executed

↓
That is why
main() is like
that design

.

→ Ex for front-end logic

Page.

Ex: `java login username password`

* THREAD SCHEDULER

→ It is another thread created by JVM, which is responsible to schedule a thread for execution.

* Every thread created in the JVM must be registered by the Thread Scheduler, so that Thread Scheduler must schedule execution.

* Whenever we call a start() method on thread, it always registers the Thread to the Thread Scheduler.

* The Thread Scheduler allocates the Stack at CPU time [resources] based on the available of resources in the system