06/07/2018
Friday.

Core Java Syllabus

Module-1 (Basics)

* Introduction
* JDK/ JRE/ JVM
* "Helloworld" program
* Keywords & Identifiers
* Operators
* Control statements (**)
* Methods (functions) (**)
* Basics of String
* Basics of Arrays

Module-2 (Oop's):

* Class and object
* Constructors
* Inheritance
* Overloading & Overriding
* Polymorphism
* Packages
* Abstract class and Abstract method
* Interfaces
* Type Casting
* Abstraction
* Java bean class
* Singleton class

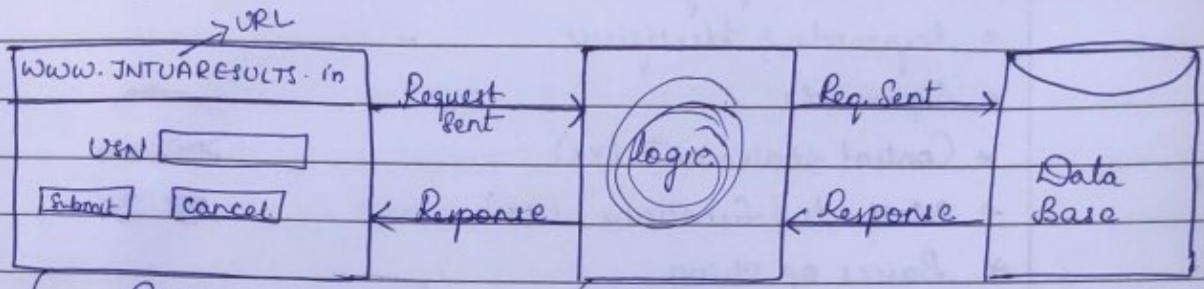
Module-3 (Libraries)

* Object class and it's methods
* String class (**)
* Exception handling (**)
* Multithreading (**)
* Collection framework library (**)
* Wrapper Classes
* File handling

__/__/____

1. Standalone Software } 5% are using Eg: Notepad, wordpad
Desktop Applications }
Doesnot require any Server.

2. Client/Server



GUI
Front end

Server
JNTUA
Middle end

Backend/
Rear end.

HTML/CSS/JS

Java/J2EE/Frameworks

Eg: FB, Insta etc.,

3. Softwares required.

a) JDK 1.8

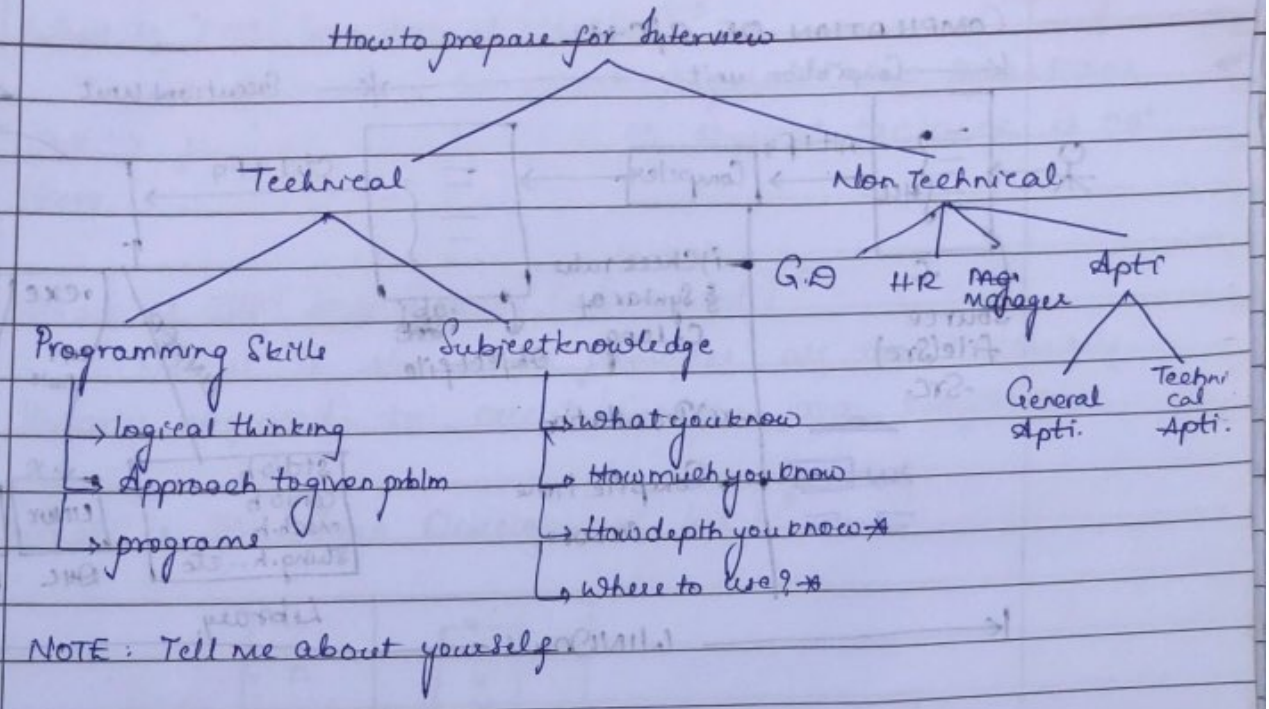
b) Editor
- Notepad
- Wordpad
- Notepad++
- Textpad
- Editplus ✓

JDK 1995 – 1.0
James Gosling – 1.1
property of – 1.2
Sun Microsystems – 1.3
– 1.4
⋮
Owned by Oracle – 1.8 ✓
– 1.9
– 1.10

c) IDE (Integrated development Environment)
- Eclipse ✓
- Netbeans

SCJP - Sun Certified Java programmer
OCJP - Oracle
OCPJP - Oracle Certified Professional Java programmer

09/07/2018.



NOTE: Tell me about yourself.

TRANSLATOR:

1. It translates one type of program into another type.

2. Translators can be 2 types:

   i) Compiler

   ii) Assembler.

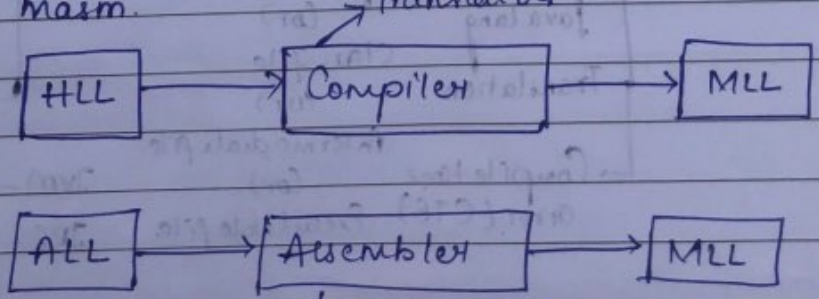
3. Compiler, translates high-level language (HLL) into machine level language where as assemblers will translate assembler level language into machine level language.

Eg. for Compiler:

Turboc, Turbocpp, Cygwin, Javac etc,

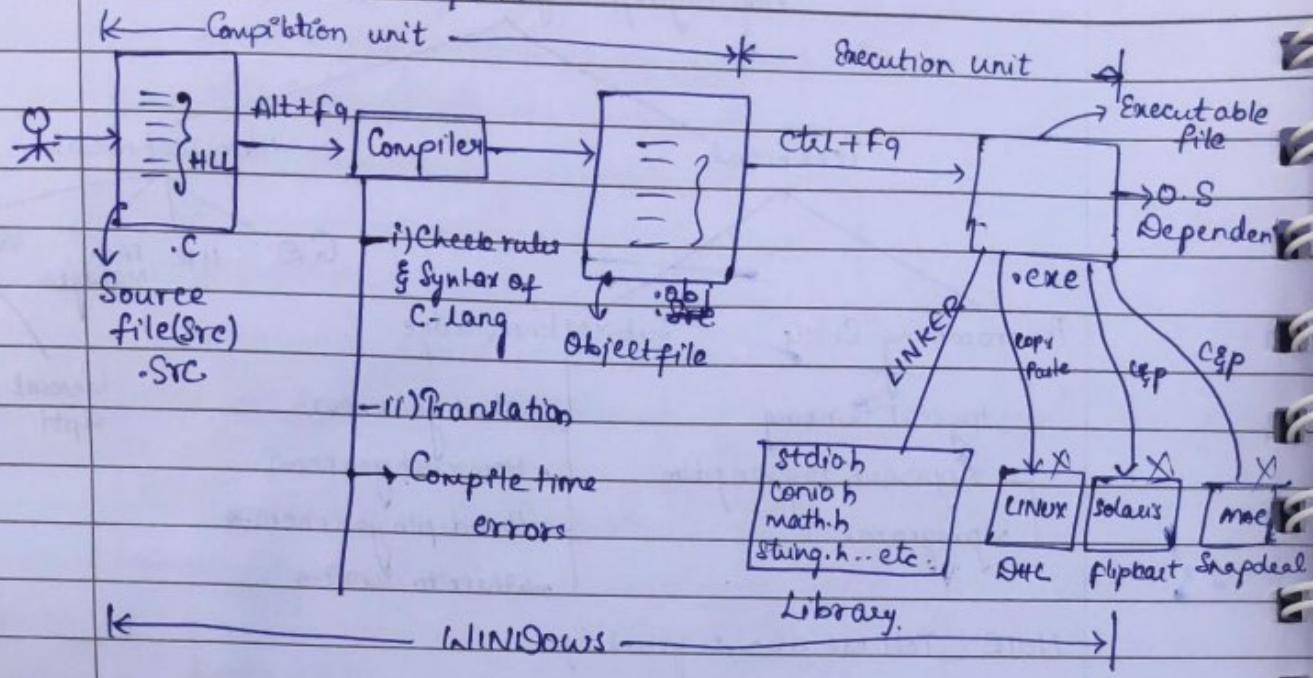
Example for Assembler:

masm.



__/__/____

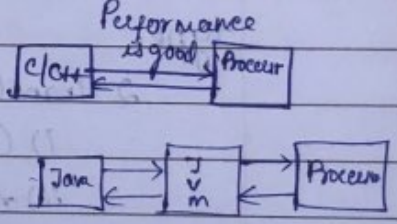
COMPILATION OF C/C++:

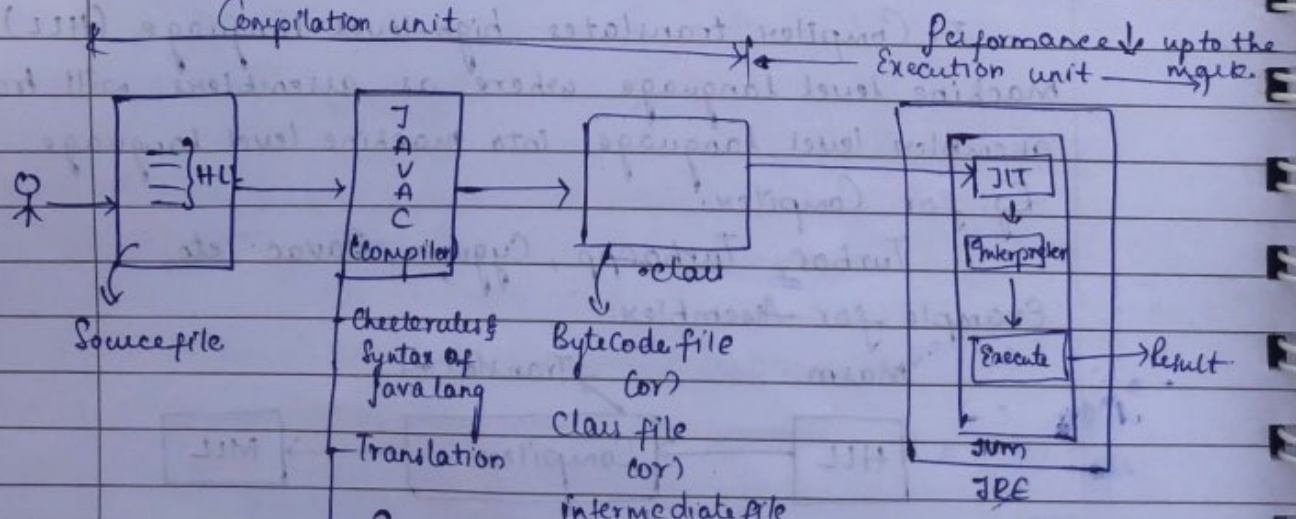


Note: Operating System / Platform / Architecture all are same.

JAVA COMPONENTS:

COMPILATION OF JAVA:





JVM – Java Virtual machine

JRE – Java runtime environment

JIT – Just in Time

10/07/2018
Tuesday.

1. What is JVM (Java Virtual Machine)?

It is a software which peforms the operations like a physical machine but its physical existance is not there.

2. What is JRE (Java Runtime Environment)?

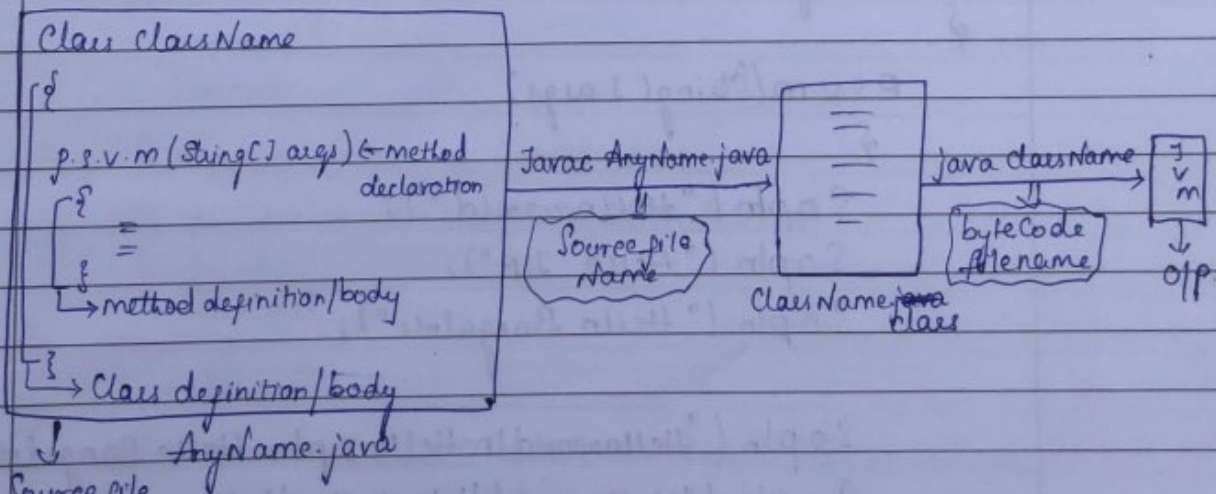
It is a s/w which provides all the necessary libraries required to execute/run the java program.

3. What is JDK (Java Development kit)?

Java development kit (JDK)

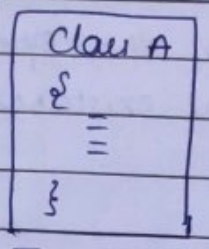
It is a software which contains the necessary components required for to compile & execute the java program.

STRUCTURE OF JAVA PROGRAM.



Compilation : Javac Sourcefile.java
Execution : java byteCode file

11/07/2018
Wednesday.

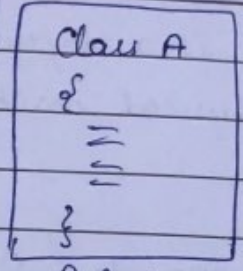


Test.java

javac Test.java

java (A)

bytecodefile



A.java

javac A.java

java (A)

byteCode file.

1. Program (Basics): program1.java.

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

javac program1.java

java Test

Output

welcome to java.

2. Program2.java

```
class Program2
{
    P.S.V.M(String[] args)
    {
        S.O.Pln("Hello world");
        S.O.Pln("Hello Jsp");
        S.O.Pln("Hello Bangalore");

        S.O.Pln("HelloworldIn HelloJspIn HelloBangalore");
        S.O.Pln("HelloworldItHelloJspIt HelloBangalore");
    }
}
```

__/__/____

Output for program2.java.

Hello world
Hello Jsp
Hello Bangalore
Hello world
Hello Jsp
Hello Bangalore
Hello world Hello Jsp Hello Bangalore.

'+' OPERATOR IN JAVA:

1. 10 + 20 -> 30
   ↳ Add
2. "Hello" + "world" => Helloworld
   ↳ Concat
3. "Hello" + 10 => Hello10
   ↳ Concat
4. 100 + "world" => 100world
   ↳ Concat
5. "Hello" + 10 + 20
   ↑ ↑
   ↳ Concat
   "Hello10" + 20 = "Hello1020"
6. 10 + 20 + "Hello" => 30 + "Hello"
   ↳ Add
   => 30 Hello.
7. 'A' + 'B' => 65 + 66 = 131 => Takes Ascii values.
8. "A" + 'B' => AB
   ↳ Concat
9. 10 + 'B' => 76 => i.e. 10 + 66 => 76.
   ↳ Add

__/__/____

"Hello" - "world" } Compile time error (CTE)
"Hello" * "world" } X
"Hello" / "world"

L → R
10+20+30

3. Program3.java

```
Class Program3
{
    P.S.V.m(String[] args)
    {
        S.o.println("Hello "+"world"); //Hello world
        S.o.pln("Hello "+"world "+10+" "+20);
                        //Hello world 10 20
        S.o.pln(10+20+"Hello "+"world");
                        //30 Hello world
        S.o.pln("Hello "+10+" world "+20);
                        //Hello 10 world 20
        S.o.pln('A'+'B');   //131
        S.o.pln('A'+10);    //75
        S.o.pln("AB"+'B');  //ABB
    }
}
```

KEYWORDS AND IDENTIFIERS:

1. Keywords are the set of predefined reserved words defined by the programming language.
2. Every programming language has its own set of keywords.
3. In java all the keywords are in lowercase.

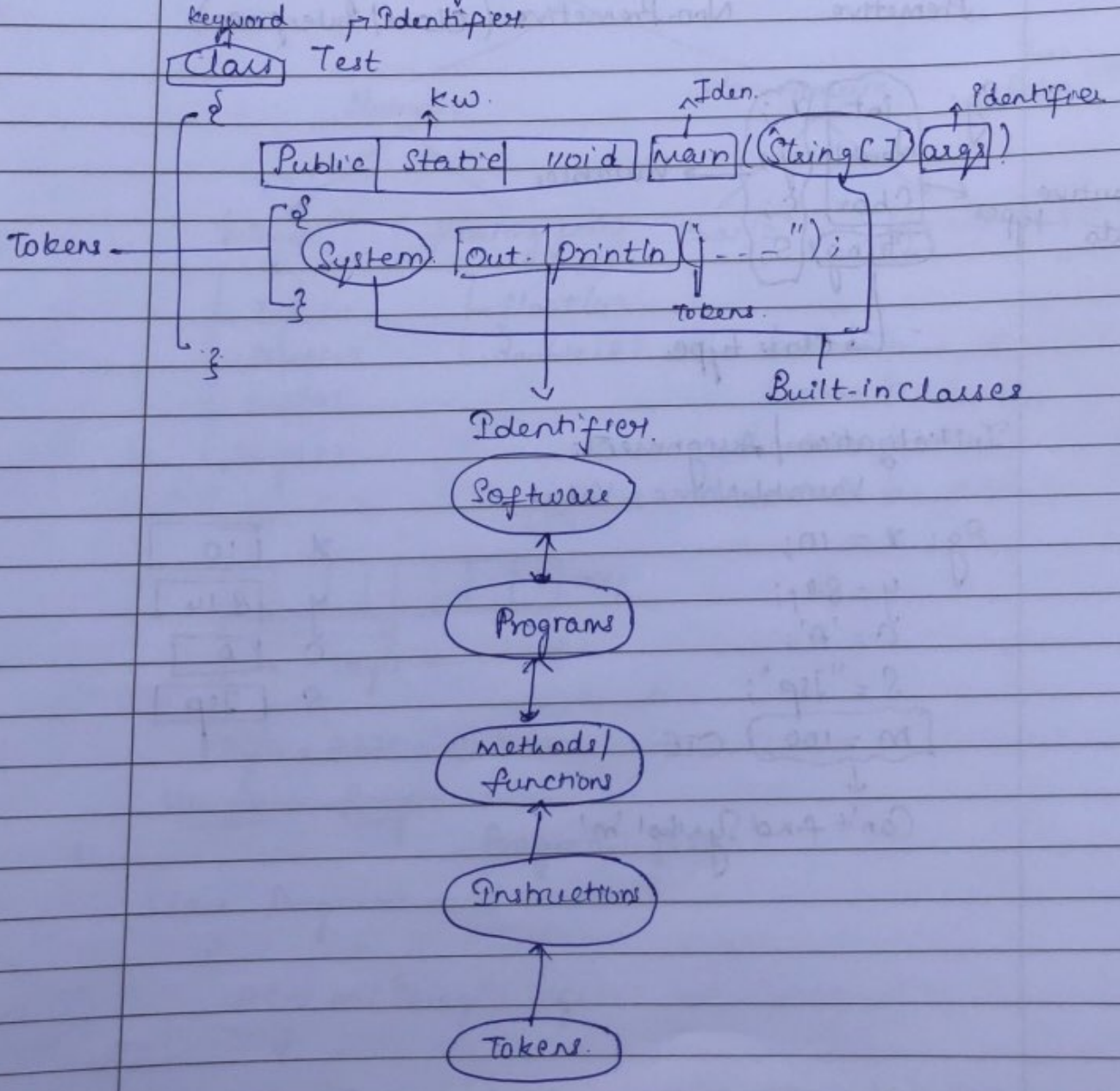
Example:

Static	boolean	public
int	true	abstract
double	long	
float	throw	

__/__/____

IDENTIFIERS:

1. They are used by the programmer to identify the elements of the program.
2. Identifiers can be one of the following:
   i) ClassName
   ii) Variables
   iii) MethodName
   iv) Package Name.



__/__/____

VARIABLES:

1. It is an identifier.
2. An identifier used to hold some value.
3. An identifier used to identify memory location.

Declaration:

DataType VariableName;

Premetive Non-Premetive (class/Interface)

Eg:

int	x;
double	y;
Char	c;
String	s;

→ Variables

Primitive data types ←

String → Class type.

Initialization / Assignment:

VariableName = Value;

Eg: x = 10;
y = 8.14;
c = 'A';
s = "Jsp";
m = 100; CTE

↓

Can't find Symbol 'm'

x	10
y	8.14
c	A
s	Jsp

12/07/2018
Thursday.

int x=10; → initialization
double y=8.14;
Char c='A';
float f=3.14f;
long l=100l;

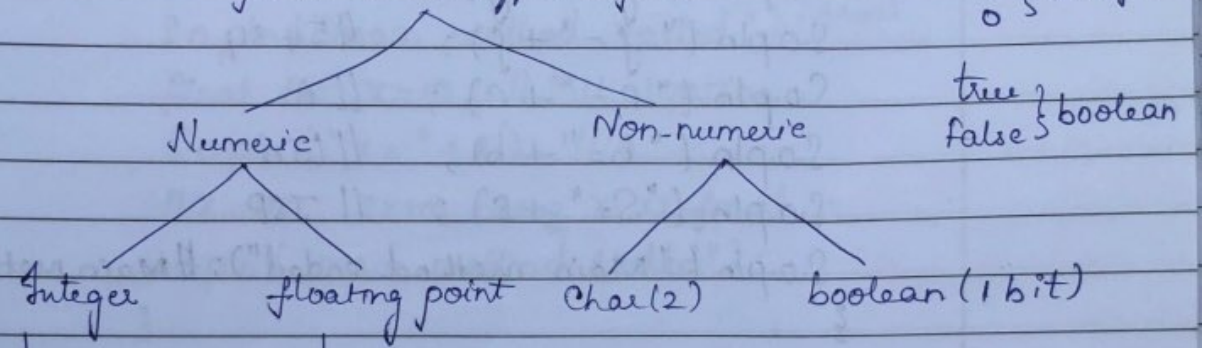
Note: By default Compiler treat decimal values as 'double', in order to say the Compiler 'float' we write 'f' in Suffix

a=1; assignment | initialized

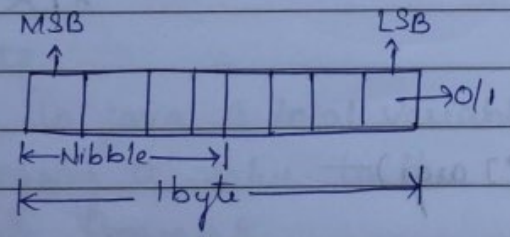
PRIMITIVE DATATYPES:

1 } C/C++
0 }
1 } x in java
0 }

true } boolean
false }



boolean b=1; X
boolean b=0; x
boolean b=true; ✓
boolean b=false; ✓



1 byte = 8 bit = 2 nibble.

Variables Programs

1. Program1.java.

```
Class Program1
{
  P.S.V.m(String[] args)
  {
    S.O.pln("Main method Started"); // Main method started
    // Initialization
```

__/__/____

```
int x=10;
double y= 323.3;
float f= 54.34f;
char c = 'A';
boolean b= true;
String S= "JSP";
S.o.pln("x = "+x);      //10
S.o.pln("y= "+y);       //323.3
S.o.pln("f= "+f);       //54.34
S.o.pln("c= "+c);       // A
S.o.pln("b= "+b);       // true
S.o.pln("S= "+S);       // JSP.
S.o.pln("Main method ended"); // Main method ended.
}
}
```

Note:

1. A variable can be re-assigned any number of times but cannot be re-declared.

2.0 Program2.java

x | ~~10~~ ~~20~~ 30

```
class Program2
{
  P.S.V.M(String[] args)
  {
    S.o.pln("main method started");
    int x=10; // Initialization
    S.o.pln("x= "+x); //10
    x=20; // Re-assignment
    S.o.pln("x= "+x); //20
    x=30; // Re-assignment
    S.o.pln("x= "+x); //30
    // int x= 40; error bcoz x already exists in main method
    S.o.p("main method ended");
  }
}
```

2 A final variable can be initialized but cannot be re-initialized.

Eg: final double pi=3.14;

2.1 Class program2
{
P.S.V.m(String[]args)
{
S.o.pln("main method started");
final int x=10; // initialization
S.o.pln("x="+x); //10
//x=20; error bcoz x is final
S.o.pln("main method ended");
}
}

int x;
S.o.p(x);
i) Junk/Garbage
ii) 0
iii) error ✓
iv) Null

x 9

final int x;
x=10; ✓

final int x=100;
x=100; ✗

Note:
In java a local variable cannot be used without assigning a value to it.

2.2 Class program2
{
P.S.V.m(String[] args)
{
S.o.pln("main method started");
int x=10,y; // initialization
y=x+y; // error bcoz 'y' is not initialized
S.o.pln("y="+y);
S.o.pln("main method ended");
}
}

✗ int x=10,y;
y=x+y; not initialized
S.o.p(y);

13/07/2018

Operators:

Operators are used in java to build the expressions.

Types Of Operators:

1. Relational Operator (<, >, =, <=, >=, !=)
2. Arithmetic Operator (+, -, /, *, %)
3. Logical Operator (||, &&)
4. Bitwise Operator (|, &)
5. Unary Operator (++, --)

1. Relational Operator:

```
1. int x=10;
   int y=20;
   S.o.p(x>y); // false
   S.o.p(x<y); // True
   S.o.p(x<=y); // True
   S.o.p(x>=y); // false
(2. int x=10;
   int y=20;)
```

2. Arithmetic Operator:

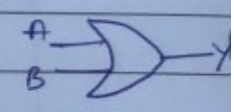
```
int x=10;
int y=20;
S.o.p(x+y);
S.o.p(x+y*y+x);
S.o.p(x+y-y+x*x);
```

AND gate

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR gate

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1



3. Logical Operator:

```
int x=10;
int y=20;
S.o.p((x>y)&&(x<y));
   (false && true) // false
S.o.p((x>y)||(x<y));
   (false || true) // true
```

__/__/____

UNARY OPERATOR:

- Post Increment
- Pre Increment
- Post Decrement
- Pre Decrement

1. int x=10;
   int y=(x++);
   S.o.p(y); //10

   x: 10 → 11

2) int x=10;
   int y=++x;
   y=11
   x=11

   x: 10 → 11

3. int y=x--;
   y=10;
   y=9

   x: 10 → 9

4. y=--x;
   y=9
   x=9

   x: 10 → 9

Examples:

```
Class Operators
{
  P.S.V.m(String[] args)
  {
    int x=10;
    int y=++x;
    S.o.pln("y= "+y);
    S.o.pln("x= "+x);

    int P=13;
    int q=P++ + ++P;
    S.o.pln("Q= "+q);
    S.o.pln("P= "+P);

    int r=29;
    int s=--r - ++r - r--;
    S.o.pln("S= "+s);
    S.o.pln("r= "+r);
```

__/__/____

CONTROL STATEMENTS:

1. Control Statements are used to divert the flow of execution of the program in java.
2. It can be achieved by using both looping as well as branching statements.

Branching
- if
- if else
- if else ladder
- Nested if
- Switch

Looping
- for
- foreach
- while
- dowhile

Examples:

1. if (Condition)
{
==
}
if

2) if (Condition)
{
==
}
else
{
==
}
if-else

3. if (Condition1)
{
==
}
else if (Condition2)
{
==
}
elseif (Condition3)
{
==
}
else
{
==
}
if else ladder

__/__/____

4. if(Condition1)
{
==
if(Condition2)
{
==
}
else
{
==
}
else
{
==
}

5. Switch(Case number)
{
case 1: ==
break;
case 2: ==
break;
case 3: ==
break;
default: ==
}

Example:

```
Class ConditionalStatements
{
public static void main (String[] args)
{
S.o.p ("Main method started");
int x=10;
int y=50;
if(x>y) // if(x<y) executes the code & print x<y.
S.o.p ("x is greater than y");
S.o.p ("Main method ended");
}
}
```

Output:

Main method Started
Main method ended.

__/__/____

2. Class ControlStatement2
```
{
P.S.V.M(String[] args)
{
S.o.p("Main method started");
double amtbal = 500.00;
double amtwithdraw = 5000.00;
if (amtbal >= amtwithdraw)
{
S.o.p("Succesful withdraw");
}
else
{
S.o.p("Maintain amount balance first");
}
S.o.p("Main method ended");
}
}
```

3. Class ControlStatement3
```
{
P.S.V.M(String[] args)
{
S.o.p("Main method started");
int pin = 1234;
double amtbal = 500.00;
double amtwithdraw = 1000.00;
if(Pin == 1234) {{
S.o.p("pin entered is Correct");
if (amtbal >= amtwithdraw)
S.o.p("Amt bal is greater than amt to be withdraw");
S.o.p("Succesful withdraw");
}
```

```
else
{
S.o.p("Maintain amt balance first");
}}
else
{
S.o.p("Incorrect pin");
}
S.o.p("main method ended");
}
}
```

4.
```
Class Control Statement4 {
P.S.V.m(String[] args){
S.o.p("main method started");
int num1=500;
int num2=500;
if(num1>num2){
S.o.p("num1 is greater");
}
elseif(num1<num2){
S.o.p("num2 is greater");
}
else{
S.o.p("num1 and num2 are equal");
}
S.o.p("main method ended");
}
}
```

__/__/__

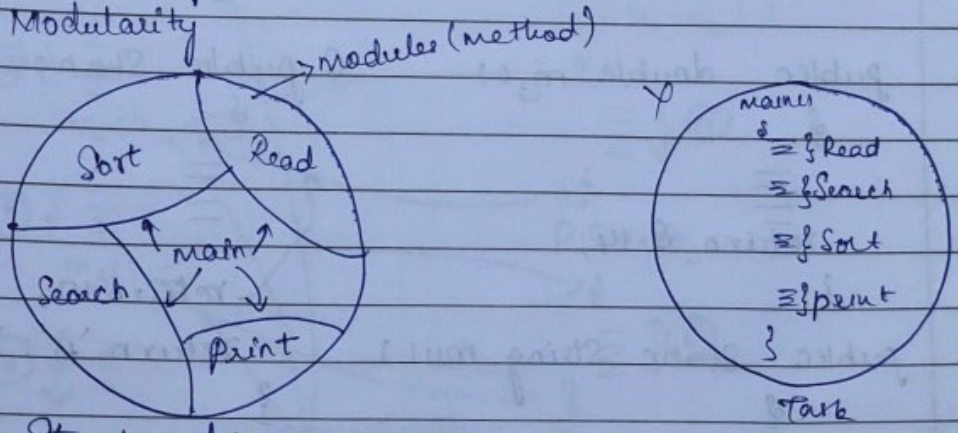
5.
```
Class ControlStatement
{
public Static Void main (String[] args)
  {
    S.o.p (" main method started");
    Switch(9)
     {
      Case 1: S.o.p ("You have got fcD");
              break;
      Case 2: S.o.p ("You have got fc");
              break;
      Case 3: S.o.p ("you have got Sc");
              break;
      default: S.o.p ("you are passed");
     }
    S.o.p ("main method ended");
  }
}
```

16/07/2018

METHODS (FUNCTIONS):

1. Methods are the set of instructions written by the user to perform some task.

2. Methods helps us in code reusability and it increases the modularity of the program.



Structured programming

Procedure Oriented programming (POP)

Un-Structured programming

Syntax: ~~Public/private/default/protected~~

<AccessModifier> <Modifier> Returntype methodname (<Argument
<Access Specifier>

↓

Static/abstract/final/Synchronized.

{

= } definition.

}

Example:

methodname

public static void main (String[].args)

{

=

}

Example

1. public static void M1()

{

=

}

__/__/____

2. Static int m2()
{
==
return 100;
}

3. public double m3()
{
==
return 8.14;
}

4. public Static String m4()
{
==
return "Hello";
}

5. public static void m5()
{
==
return;
}

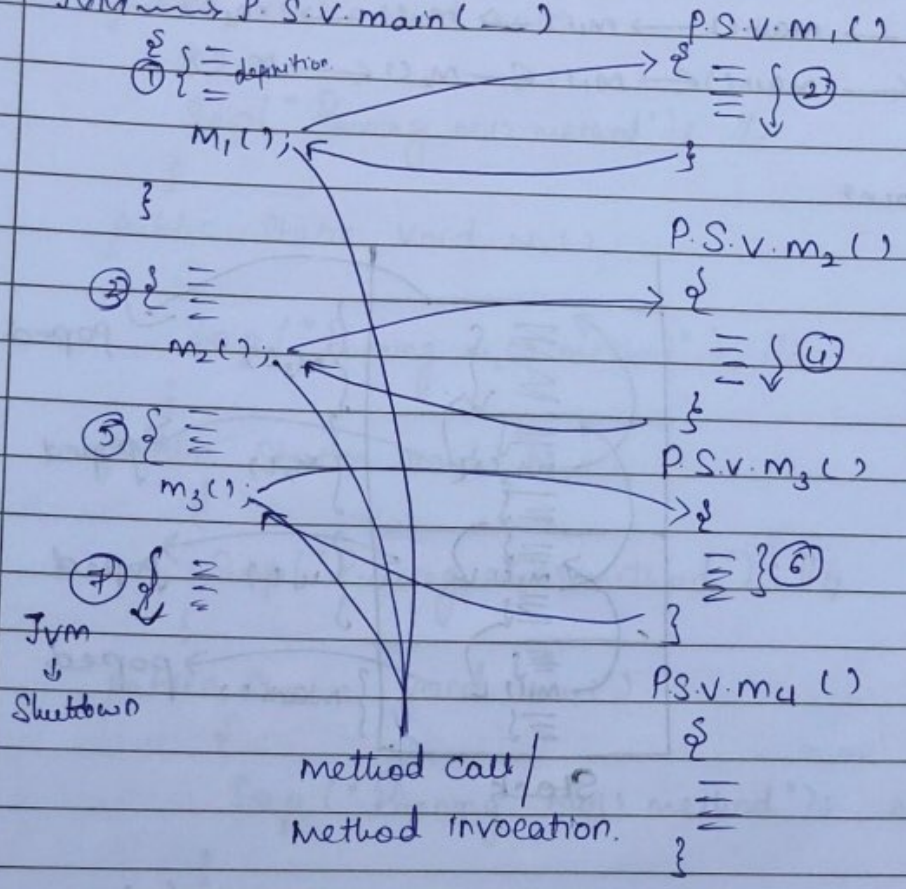
6 public Static void m6()
{
==
return 10;
return 'A'; XCTE
}

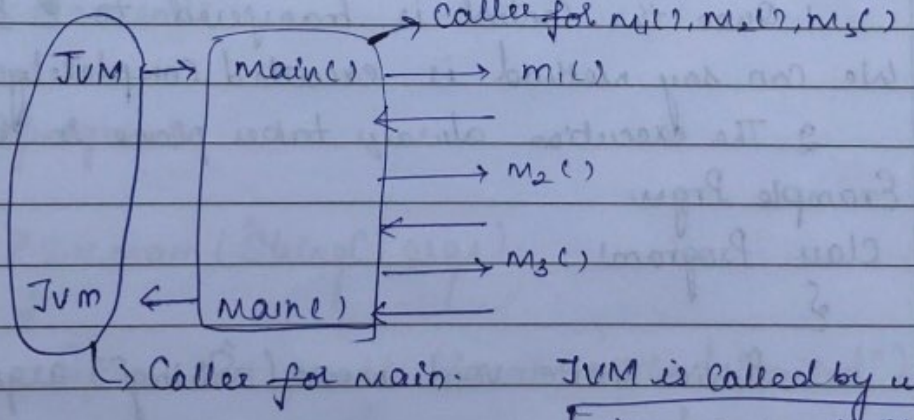
7. public Static boolean m7()
{
if (10>20)
{
return true;
}
else
{
return false;
}
}

JVM - Caller

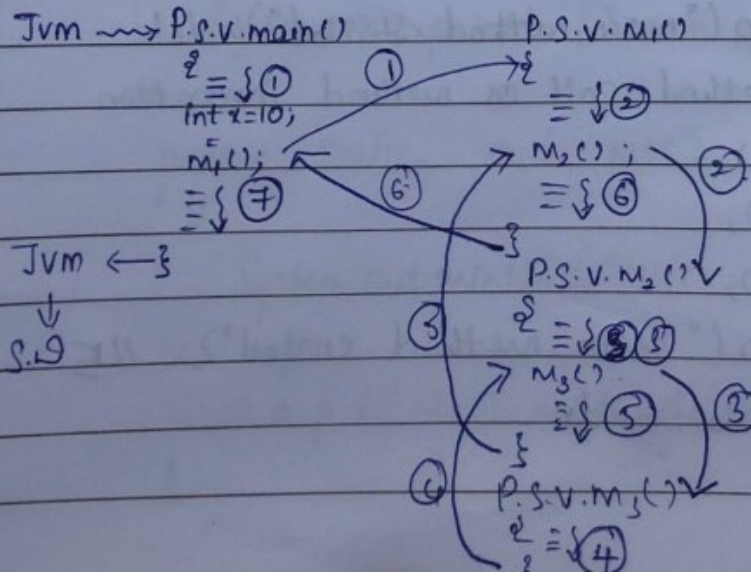
17/07/18

8) JVM ⇝ P.S.V.main(---)





JVM is called by using java bytecode file



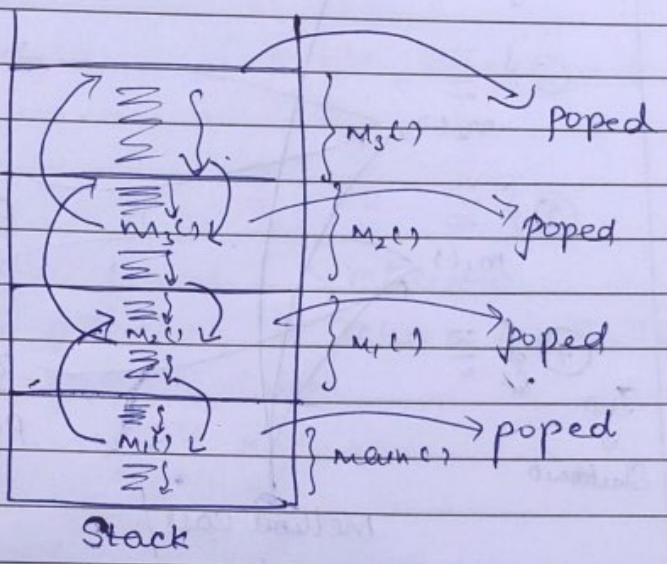
__/__/____

JVM → main() → $M_1()$ → $M_2()$ → $M_3()$

JVM ← main() ← $M_1()$ ← $M_2()$ ← $M_3()$

↓

Shutdown



Stack

Note:

1. Once the control is transfered to } (or) return then we can say method is executed completely.
2. The execution always takes place in stack

Example Prgm

```
Class Program1
{
    Public Static void main (String[] args)
    {
        S.o.p("main method started"); //1
        // Method call or method invocation
        M1();
        M2();
        M3();
        S.o.p("main method ended"); //5
    } → main ends
```

__/__/____

```
public static void M1()
  {
   S.o.p("Running M1() method"); //2
  }
public Static void M2()
  {
   S.o.p("Running M2() method"); //3
  }
public Static void M3()
  {
   S.o.p("Running M3() method"); //4
  }
public Static void M4()
  {
   S.o.p("Running M4() method"); no error
  }
}→ class ends.
```

program.2:

```
Class Programe2
{
  P.S.V.main(String[] args)
  {
   System.out.println("main method Started"); (1)
   M1();
   System.out.println("main method ended"); (8)
  }
  public Static void M1()
  {
   System.out.println("M1() method Started"); (2)
   M2();
   S.o.p("M1() method ended"); (7)
  }
```

__/__/____

```
public static void m2()
{
    S.o.p("m2() method started"); (3)
    m3();
    S.o.p("m2() method ended"); (6)
}
public static void m3()
{
    S.o.p("m3() method started"); (4)
    S.o.p("m3() method ended"); (5)
}
}
```

PASSING AND RETURNING ARGUMENTS TO/FROM A METHOD.

1) m1();

```
P.S.V.m1()
{
}
```

Passing nothing returning nothing.

2) m2

```
double y = m2(10);
S.o.p(y); //51.4
P.S.double m1(int x)
{
    S.o.p(x); //10
    return 8.14;
}
```

Passing something returning something

3) `m3('A');`

```
P.S.V. m3(Char ch)
{
    S.o.p(ch); //A   Passing Smthg returning nthg.
    {
    }
}
```

4) `String Str = M4();`
`S.o.p(Str);`

```
P.S. String M4()  passing nthg return Smthg.
{
    {
    return "Jsp";
}
```

5)
```
P.S.V. M1()
{
    {
        M1();
    }
}
```
Recursion

We can Call a method inside another method

6)
```
P.S.V. M1()
{
    P.S.V. M2()
    {
    }
}
```

But we Cannot Call the execute a method inside a another method.

✗ Not valid.

__/__/____

program3.

```
Class Program3
{
    public static void main(String[] args)
    {
        S.o.p("main method started");
        int num1=10, num2=5;

        int Sum = addTwoNum(num1, num2); //method call
        int Sub = SubTwoNum(num1, num2);
        int prod = mulTwoNum(num1, num2);
        int div = divTwoNum(num1, num2);

        S.o.p("Sum of "+num1+"&"+num2+" = "+Sum);
        S.o.p("Sub of "+num1+"&"+num2+" = "+Sub);
        S.o.p("mul of "+num1+"&"+num2+" = "+prod);
        S.o.p("div of "+num1+"&"+num2+" = "+div);

        S.o.p("main method ended");
    }
    P. S. int addTwoNum(int num1, int num2)
    {
        return num1+num2;
    }
    P. S. int SubTwoNum(int num1, int num2)
    {
        return num1-num2;
    }
    P. S. int mulTwoNum(int num1, int num2)
    {
        return num1*num2;
    }
```

18/07/2018
Wednesday

```
P.S. int divTwoNum(int num1, int num2)
    {
        return num1/num2;
    }
}
```

Assignments:

1. Write a program to check whether number is odd or even.
2. Write a program to check biggest of two numbers.
3. Write a program to check whether the number is +ve or -ve.

READING INPUTS FROM THE USER:

Step-1:

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

Step-2:

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

Step-3:

```
int    => sc.nextInt();
double => sc.nextDouble();
long   => sc.nextLong();
short  => sc.nextShort();
char   => sc.nextChar();   x => sc.next().charAt(0);
String => sc.next(); //Reads upto space
String => sc.nextLine(); //Reads Complete line
```

Example.

```
import java.util.Scanner;
class program5
{
    P.S.V.M(String[] args)
    {
        S.o.p("main method started");
```

```
Scanner sc = new Scanner(System.in);
S.o.p("Enter the college name");
String cname = sc.nextLine();
S.o.p("Enter ur name");
String sname = sc.next();
System.out.println("Enter the id");
int sid = sc.nextInt();
S.o.p("Enter the marks");
double smarks = sc.nextDouble();
S.o.p("Enter the grade");
char sgrade = sc.next().charAt(0);
S.o.p("Student details are");
S.o.p("Name = "+sname);
S.o.p("Id = "+sid);
S.o.p("marks = "+smarks);
S.o.p("Grade = "+sgrade);
S.o.p("College Name = "+cname);
S.o.p("main method ended");
}
}
```

2

```
Import java.util.Scanner;
Class Program6
{
    public static void main(String[] args)
    {
        S.o.p("main method started");
        Scanner sc = new Scanner(System.in);
        S.o.p("Enter the first number");
        int num1 = sc.nextInt();
        S.o.p("Enter the second number");
        int num2 = sc.nextInt();
        findBiggest(num1, num2); // method call
        S.o.p("main method ended");
    }
```

__/__/____

```
public static void findBiggest ( int num1, int num2)
 {
   if (num1 > num2)
     {
       S.o.p(num1+ " is greater than " +num2);
     }
   else if (num1 < num2)
     {
       S.o.p(num1+" is less than "+num2);
     }
   else
     {
       S.o.p(num1+" is equal to "+num2);
     }
 }
}
import java.util.Scanner;
 class Program7
  {
    P.S.V.m(String[] args)
     {
       S.o.p("main method started");
       Scanner sc= new Scanner(System.in);
       S.o.p("Enter the String");
       String str = sc.nextLine();
      forwardPrint(str); //method call
      ~~for~~ backwardPrint(str); // method call
       S.o.p("main method ended");
     }
    public static void forwardPrint (String s)
      {
        S.o.p("forward Direction");
```

__/__/____

```
S.o.p("_____");
for (int i=0; i<S.length(); i++)
{
    S.o.p(S.charAt(i));
}
```

```
int i=0;
while (i<S.length())
{
    S.o.p(S.charAt(i));
    i++;
}
```

```
public static void backwardPrint(String s)
{
    S.o.p("Backward direction");
    for (int i=S.length()-1; i>=0; i--)
    {
        S.o.p(S.charAt(i));
    }
}
```

Assignment:

1. Read a string from the user and print half of the characters.

2. Read a String from the user and check for specific character existence.

3. Read a String from the user and count the no of occurance of a specific character.

Harshitha

a Honey

i=0 i<-1

j=0 i<

Honey

i=0

j=0 1 0<5

Hello

for(i=0; i<S.length(); i++)

for(j=1; j<S.length(); j++)

{

Codility
hacker.
hackerRank.

19/07/18

4. Read a number from a user and reverese it.

```
int rev=0
while (num>0)
{
  rem = num%10;
  rev = rev*10 + rem
  num = num/10;
}
```

①	②	③
rem = 6	rem = 4	rem = 8
rev = 6	rev = 64	rev = 648
num = 84	num = 8	num = 0

1. Read the string from the user and extract the Substring from it.

Read string, Read starting position, Read ending position

```
public static String extractSubString(String S, int SP, int EP)
{
}
```

20/07/2018.
Friday.

2

SP >= 0 && SP <= S.length()-1 && (SP <= EP)

T T T

0 <= SP <= S.length()-1

Hello

0 >= 4

0

20/07/2018

Loops:

1)
```
Class Loop_prgs
{
  P.S.V.M(String[] args)
  {
    for(int i=0; i<5; i++)
    {
      S.o.p(i);
    }
  }
}
```

for Syntax:

```
for(initialization; condition; inc/dec)
{
  ≡
  body
  ≡
}
```

```
for(int i=0; i<5; i++)
{
  S.o.p(i);
}
```

O/P:

0 1 2 3 4

2)
```
for(int i=0; i<=5; i++)
{
  for(int j=0; j<=5; j++)
  {
    S.o.p(i);
  }
  S.o.println();
}
```

0 0 0 0 0 0
1 1 1 1 1 1
2 2 2 2 2 2
3 3 3 3 3 3
4 4 4 4 4 4
5 5 5 5 5 5

3)
```
for(int i=0; i<5; i++)
{
  for(int j=0; j<=i; j++)
  {
    S.o.p(i); // S.o.p(j)
  }
  S.o.p();
}
```

o/p:

```
0
1 1
2 2 2
3 3 3 3
4 4 4 4 4
```

```
0
0 1
0 1 2
0 1 2 3
0 1 2 3 4
```

4
```
for(int i=5; i>0; i--)
{
  for(int j=5; j>0; j--)
  {
    S.o.p(i);
  }
}
```

```
5 4 3 2 1
5 4 3 2 1
5 4 3 2 1
5 4 3 2 1
5 4 3 2 1
```

5.
```
for(int i=5; i>0; i--)
{
  for(int j=5; j>=i; j--)
  {
    S.o.p(i);
  }
}
```

o/p:

```
5
4 4
3 3 3
2 2 2 2
1 1 1 1 1
```

6)
```
for(int i=5; i>0; i--)
{
  for(int j=1; j<=5; j++)
  {
    S.o.p(i);
  }
```

```
5 5 5 5 5
4 4 4 4
3 3 3
2 2
1
```

__/__/____

7. `for (int i=0; i<=5; i++); // ERROR.`
```
{
S.o.p(i);
}
```
8. `for (int i=0; i<=5; i++)`
`S.o.p(i);`
9. `for (int i=0; i<5; i++)`
```
{ S.o.p(i);
S.o.p(i+1);
}
```
10. `int i=0;`
`for ( ; i<=5; i++)`
```
{
S.o.p(i);
```
11. `int i=0;`
`for ( ; i<=5; )`
```
{
S.o.p(i);
i++;
}
```

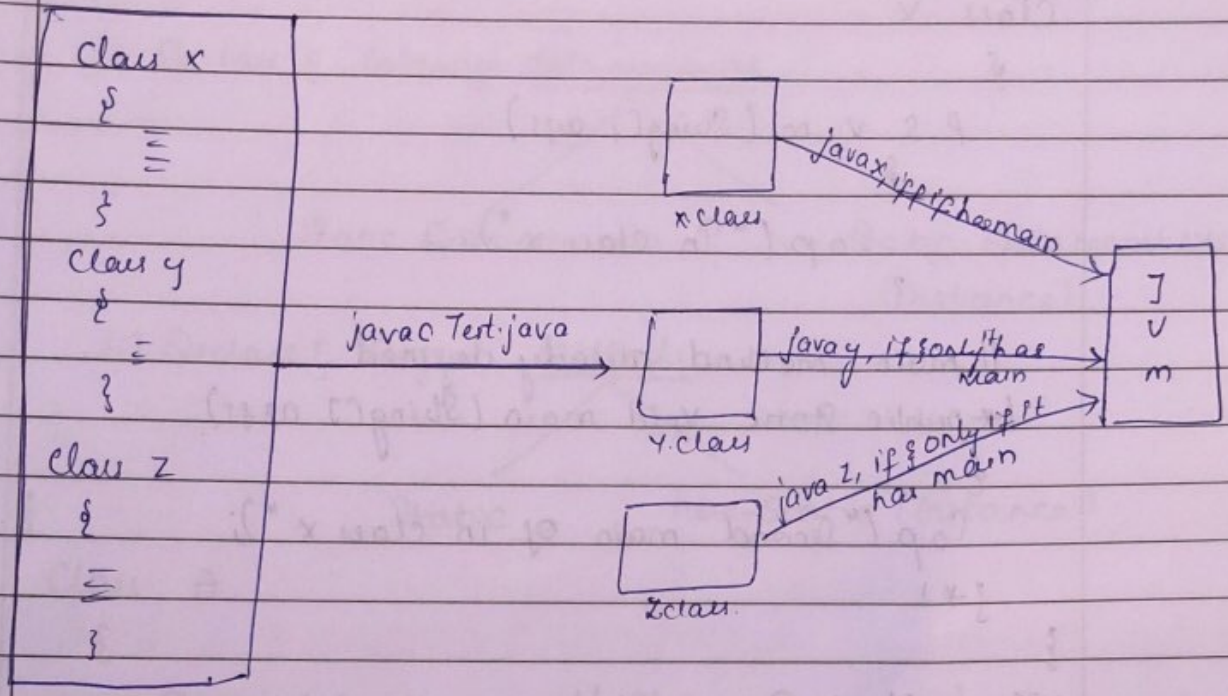
While loop.

```
while (Condition)        do
{                        {
  ==                       == incldec;
  incldec;                 } while(Condition);
}
```

1. While loop checks the condition and executes its body.
1. for the first time Do body will be executed and then it checks for the condition If that given condition is satisfied body of the do will be executed once again.

23/07/2018

Section-2



public static void main(String[] args)

Swap

Runtime error
⇓
no main method found

1. A source file can contain any no. of classes (but it is not recommanded).
2. The no. of byte code file generated is equal to no. of classes present in the source file.
3. for the JVM to execute we should give only that byte code file which has a main method according to the below standard.

public static void main(String[] args)

Swapped

4. If the byte code file doesn't contain main method we will get a runtime error saying no main method found.

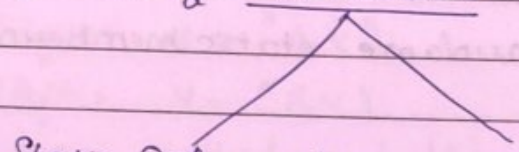
__/__/__

Oops-basics.

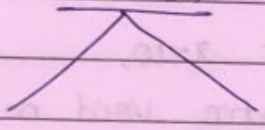
```
Class X
  {
    P.S.V.m(String[] args)
      {
        S.o.p("In class x");
      }
  // main method already defined
  /*public static void main(String[] args)
   {
    S.o.p("Second main of in class x");
   }*/
}
Class Y
  {
    Static public void main(String[] args)
      {
        S.o.p("In class y");
      }
  }
Class Z
   {
     Public static void main(String[] args)
        {
          S.o.p("In class z");
        }
   }
```

__/__/____

Class ClassName
{
i) Declare & Initialize data members



ii) Declare & Define methods



}

Class A
{
Static int x=10; // Static D.M
int Y=20; // nonstatic Dm
} ~~Data members~~

public static void m1()
{
int m=100;
// static method
}
public void m2()
{
int n=200;
Static int p=100; X
// non static method
}
}

Local variable.

1. Local variable
2. Static variable
3. Non-Static variable

__/__/____

Accessing Static members of a class:

1. Static members of the class can be accessed by using the classname

Syntax: ClassName.staticmemberName

Example:

```
Class A
{
    Static int x=10;
    public static void m1()
    {
        S.o.p("In A class");
    }
}
Class B
{
    Static int y=20;
    public static void m2()
    {
        S.o.p("In B class");
    }
}
Class Program1
{
    public static void main(String[] args)
    {
        System.out.println("main method started");
        S.o.p("Before x="+A.x);
        S.o.p("Before y="+B.y);
        A.m1();
        B.m2();
    }
}
```

O/p:

```
        A.x=100;
        B.y=200;
        S.o.p("After x = "A.x);
        S.o.p("After y = "B.y);
        S.o.p("main method ended");
    }
}
```

Note:

1. Static, non-Static and local variable can be re-assigned any no. of times as long as they are not final.
2. Static & non-static data members by default contains default values.

Eg: int - 0, double 0.0, String null.

Program2.

```
class Test
{
    Static double p=123.21;
    public static void print()
    {
        S.o.pln("Running print() method");
    }
}
class Demo
{
    public static void disp()
    {
        System.out.println("p= "+Test.p);
        Test.print();
    }
}
```

__/__/____

```
Class Program2
{
    public static void main(String[] args)
    {
        S.o.p("main method started");
        Demo.disp();
        S.o.p("main method ended");
    }
}
```

O/p:

```
main method started
P=123.21
Running Print() method
main method ended.
```

Program3.java

```
Class Calculator
{
    public static int add(int x, int y)
    {
        return x+y;
    }
    public static int sub(int x, int y)
    {
        return x-y;
    }
    public static int mul(int x, int y)
    {
        return x*y;
    }
```

```
public static int div(int x, int y)
    {
      return x/y;
    }
}
class Program3
{
  public static void main(String[] args)
  {
    S.o.p("main method started");
    int num1=20, num2=5;
    S.o.p("Sum of"+num1+"&"+num2+"="+Calculator.add
                                    (num1,num2));
S.o.p("Sub"="Calculator.sub(num1, num2));
S.o.p("mult="Calculator.mul(num1, num2));
S.o.p("div="Calculator.div(num1, num2)); S.o.p("main method
                                              ended");
    }
}
```

Assignments:

1. Write a prgm to find the biggest of 2 num, read the 2 numbers from the user. [ ] [ ] → logic with class
2. Read a number from the user and check whether it is odd or even

```
public s. void inOddEven(_)
```

24/07/2018
Tuesday.

ACCESSING NON-STATIC MEMBERS:

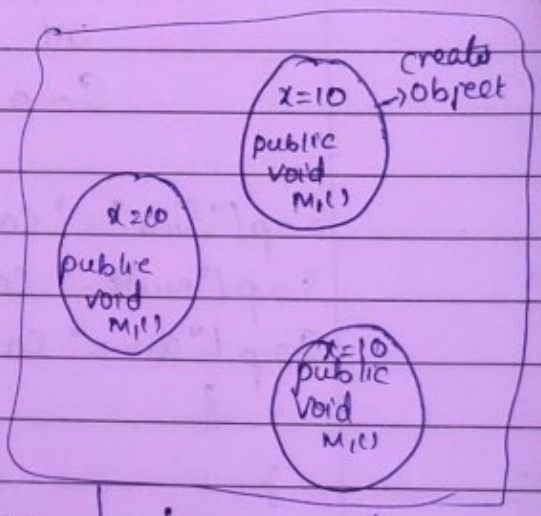
```
Class Test
{
    int x=10;
  public void M1()
    {
      =
    }
}
```

Step-1:

Create an Object by using "new" operator.

`new className();`

Eg: new Demo();
new A();
new Sample();
new Test();
new Test();
new Test();



Step-2.

Create reference variable and make it to refer to created Object.

Demo d;
A a; → Reference Variable.
Sample s;

Test t; // t is of Test type
t is a non-premitive type
t is a class type (also Interface type)

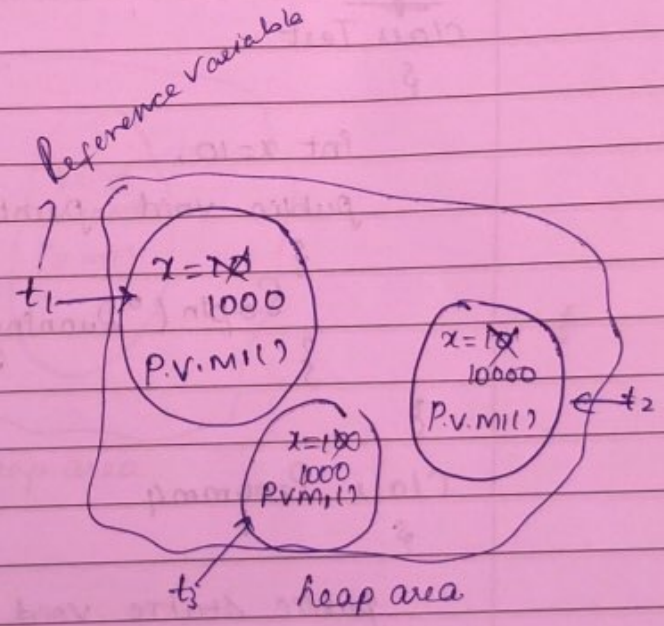
Example:

```
class Test
{
    int x=10;
    public void m1()
    {
      ≡
    }
}
```

1. Creates an object
2. Search for non-static members Load

```
Test t1,t2,t3;
t1=10;
t2=8.14;   } Error.
t3='A';
```

```
t1= new Test();
t2= new Test();
t3= new Test();
S.o.p(t1.x);
S.o.p(t2.x);   } 10
S.o.p(t3.x);
t1.m1();
t2.m1();   } ✓
t3.m1();
```




```
t2.x=100;
t1.x=1000;     } Re-assignment
t3.x=10000;
S.o.p(t1.x); //1000
S.o.p(t2.x); //100
S.o.p(t3.x); //10000
```

Instantiation:

1. It is a process of creating an object of a class. If we want to access non-static members of the class we need to instantiate the class.

Creates object

```
Test t = new Test();
```

Class Name — Reference Variable — keyword — Constructor

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

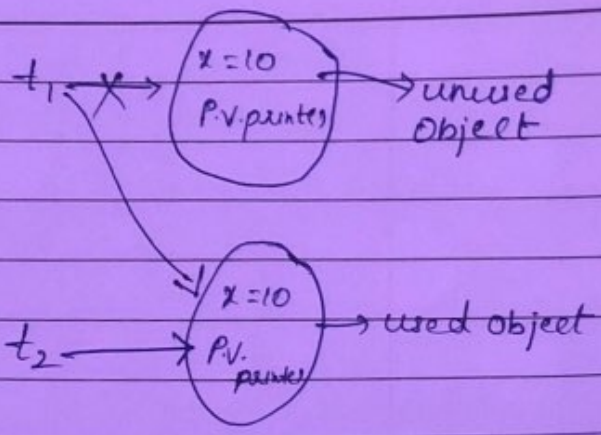
parameterized Constructor

This is non static.

Program 4.

```
class Test
{
    int x=10;
    public void print()
    {
        S.o.pln("Running Print() method");
    }
}
class Program4
{
    public static void main(String[] args)
    {
        S.o.pln("Main method started");
        /* Test t = new Test();
        S.o.p("x= "+t.x);
        t.print(); */
        S.o.p("x= "+ new Test().x);
        new Test().print();
        S.o.p("main method ended");
    }
}
```

Test t_1 = new Test();
Test t_2 = new Test();
$t_1 = t_2$

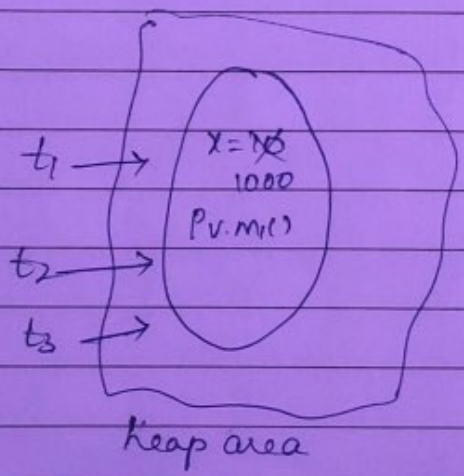


Note:

At a time one Reference variable cannot refer to two objects.

Another example

$t_2 = t_1$;
$t_3 = t_2$;



25/07/2018
Wednesday.

Static & Non-Static

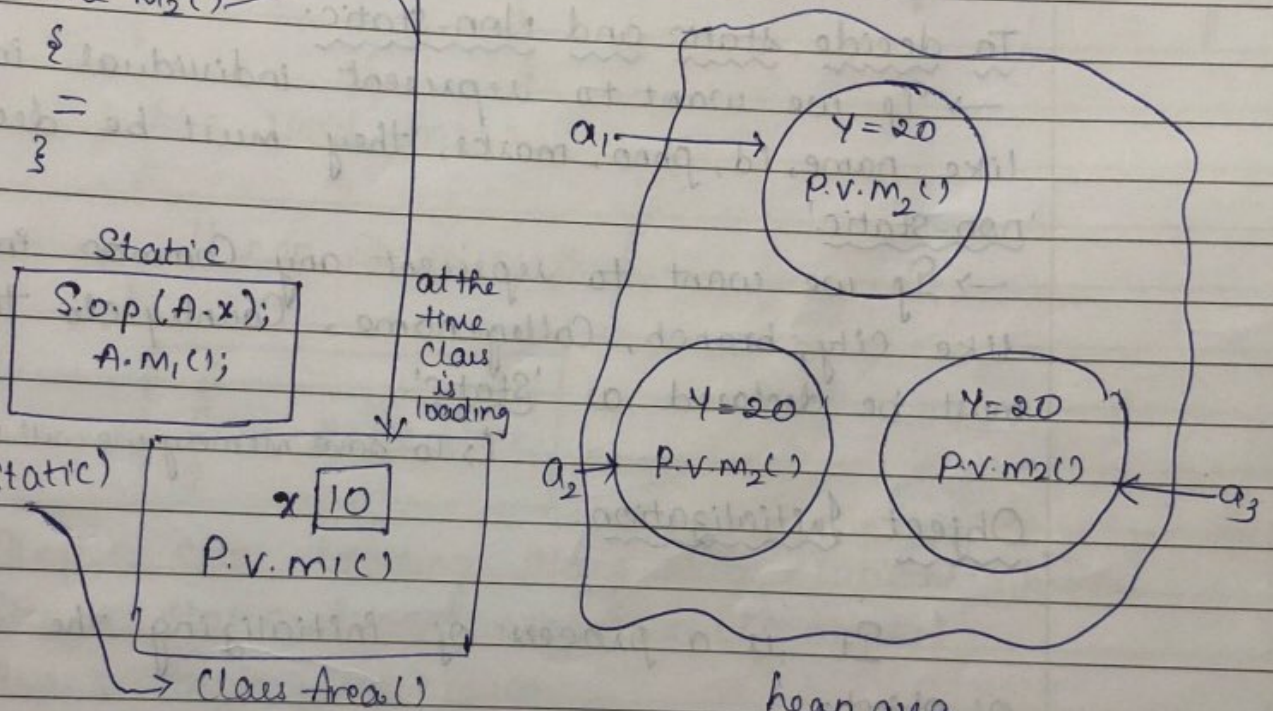
```
class A
{
    static int x=10;
        int y=20;
    static void M1()
    {
    =
    }
    void M2()
    {
    =
    }
}
```

Non-Static (Create Object)

A a1 = new A();
A a2 = new A();
A a3 = new A();

(non-static)

at the time of object creation



Note:

1. Static Members of class will be loaded at the time of class loading where as non-static Members of class will be loaded at the time of Object creation.
2. Only one copy of the static members will be loaded at the time of class loading where as no. of non-static members copy loaded depends upon Object created.
3. Static members belongs to the entire class non-static members belongs to the individual Objects.

4. Class loading activity takes place first then the object creation activity.

```
Class Test
{
  S.o.p ("Hello");
}
```
} * error.

To decide static and Non-static:

→ If we want to represent individual information like name, id, phno, marks, they must be declared as 'non-static'

→ If we want to represent any common information like city, branch, Collegename, Country etc then they must be declared as 'Static'.

↳ To save memory we will use static.

Object Initialization:

It is a process of initializing the data members of object.

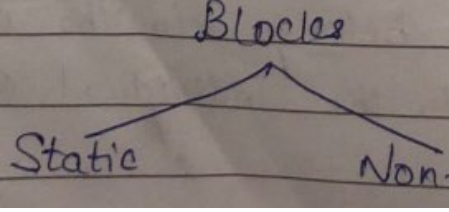
Object can be initialized by using the following two ways.

1) Instance block (Non-static block) (rare)
2) Constructor *

↳ Purpose

To use object initialize

Blocks:



```
class Test
{
  public static void m1()
  {
    // Static method
  }
```

```
Static
{
  // Static block
}
```

will be executed automatically at the time of class loading.

```
  public void m2()
  {
    // non-Static method
  }
```

```
{
//non-Static block
}
```

Step-1: Class loading along with static members.
Step-2: Static block execution (if exists)
Step-3: Begin from main
Step-4: Object creation in heap memory.
Step-5: Non-Static members loading into the object.
Step-6: Non-static block execution (if exists)
Step-7: Constructor call.

```
Class Program1
{
  Static
  {
    S.o.p ("Static blocks executed"); //1
  }
```

```
{
System.out.println("Non-Static block executed"); //3
}
public static void main(String[] args)
{
S.o.p("main method started"); //2
new Program1();
S.o.p("main method ended"); //4
}
}
```

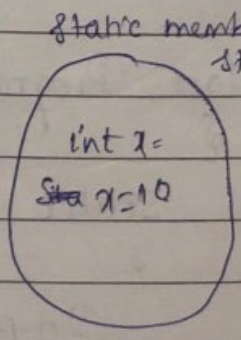
Note:

* A class can Contain any no of static blocks & non-Static blocks.
* But first executes all the static blocks after that execute all non-static blocks.

Diff b/w block and a method.

Block	Method
1. It does not have any name.	1. It will have some name
2. It will not have any return type.	2. It will have some return type
3. Will be executed automatically	3. Must be called Explicitly.

```
Class Program2
{
    Static Int x;
           Int y;

    Static
    {
      x=10;
      //y=20; // error
    }
```



26/07/2018

```
{
  x = 100;
  y = 200;
}
public static void main (String[] args)
{
  S.o.p ("Main method started");
  S.o.p ("Before Creating Obj = " + program2.x); //10
  program2 P = new Program2();
  S.o.p ("After Creating obj = " + program2.x); //100
  S.o.p ("y = " + P.y); //200
  S.o.p ("Main method ended");
}
}
```

CONSTRUCTOR:

1. It is a special member function of a class which will have be same name as that of the class.
2. Constructors are used to initialize an object.
3. Constructors will be called automatically when we create an object.
4. Hence they don't have any return type.
5. In java, every class will have either default Constructor (or) user defined Constructor but not both.
6. User defined Constructor can be no argument (or) argument. Where as default Constructor is always no argument.

Syntax:

```
<Accessmodifier> ClassName (<Arguments>)
   optional                 optional
{
  =
  =
  =
}
```

__/__/____

Example:

```
Clas Test
{
    Test()
    {
      S.o.p("Running Constructor");
    }
}
new Test();
```

user defined no-argument Constructor.

```
Clas Demo
{
  // Default Constructor;
}
new Demo()
```

Testpot.com

Note:

1. All no-argument Constructors are not default Constructors. Where as all default constructors are no arguments.

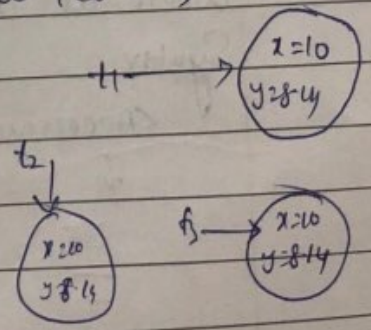
```
Clas Test
{
  int x;
  int y;
  Test()
  {
    x=10;
    y=8.14;
  }
}
```

user defined No-arg Constructor

Test t_1 = new Test();

Test t_2 = new Test();

Test t_3 = new Test();



As shown above, if a class Contains only no-argument constructor then all the objects will have same values. In Order to have the different values in different objects then we will go for argument Constructor. (parameterized Constructor).

```
Class Test
{
  int x, y;
  Test(int arg1, int arg2)
  {
    x=arg1;
    y=arg2;
  }
}
```

user defined Argument Constructor.

```
Test t  = new Test();  X
Test t1 = new Test(10,20);
Test t2 = new Test(30,40);
Test t3 = new Test(50,60);
Test t4 = new Test(10);
Test t5 = new Test(10,20,30);
Test t6 = new Test(10,10.11);
```

t1 → x=10, y=20
t2 → x=30, y=40
t3 → x=50, y=60

As shown above if a class Contains only argument Constructor then it allow us to create an Object only if we pass right number of arguments and right types of arguments.

If we want to create an object with different length of arguments, type of arguments and order of occurance of arguments then we should overload the Constructor.

Constructor Overloading:

Writing multiple constructor with different length of arguments, type of arguments and order of occurance of arguments is known as "Constructor Overloading".

Example

```
Class Test
{
    int x;
    double y;
    Test()
    {
        x=0;
        y=0.0;
    }
    Test(int arg)
    {
        x=arg;
    }
    Test(double arg)
    {
        y=arg;
    }
    Test(int arg1, double arg2)
    {
        x=arg1;
        y=arg2;
    }
    Test(double arg1, int arg2)
    {
        x=arg2;
        y=arg1;
    }
}
```

Constructor Overloading

```
new Test();
new Test(10);
new Test(10.11);
new Test(10, 10.11);
new Test(10.11, 10);
new Test(10,20,36); X
```

__/__/__

1. Create a class called Student with the attributes (datamembers) name, id and marks. Initialize the Student object through the constructor.

```
Class Student
{
    String name;
    int id;
    double marks;
    Student ( String arg1, int arg2, double arg3)
    {
        name = arg1;
        id = arg2;
        marks = arg3;
    }
}
Class Program1
{
    p s v m ( String[] args)
    {
        Student S1 = new Student ( "Onga", 10, 65.45);
        Student S2 = new Student ( "Dingi", 20, 75.32);
        S.o.p ("First Student details");
        S.o.p ("Name = " + S1.name);
        S.o.p ("Id = " + S1.id);
        S.o.p ("Marks = " + S1.marks);
        S.o.p ();
        S.o.p ("Second Student details");
        S.o.p ("Name = " + S2.name);
        S.o.p ("Id = " + S2.id);
        S.o.p ("marks = " + S2.marks);
    }
}
```

2. Create a class called Car with the attribute name, Color and prize. Create multiple objects of the car and initialize through constructor. The program should be able to create an objects with different type of arguments and different types of length of arguments.

3. Create a class called mobile with the attributes name, prize and color. Initialize the mobile class through constructor. The program should be able to create mobile objects with different length of arguments and type of arguments.

Important Conclusion:

Case-1:

Constructor don't accept any return type. If we give returntype to a constructor, we will not get any error or exception instead a constructor will be converted into a method which needs to be called explicitly just like any other method.

Ex:

```
Class A
  {
    void A()
      {
        S.o.p("Running Constructor");
      }
  }
A a = new A();
a.A(); → Method call
```

This class has default Constructor.

__/__/____

CASE-2:

Constructors don't accept any modifier. That is constructor can not be a static or final or abstract or Synchronized.

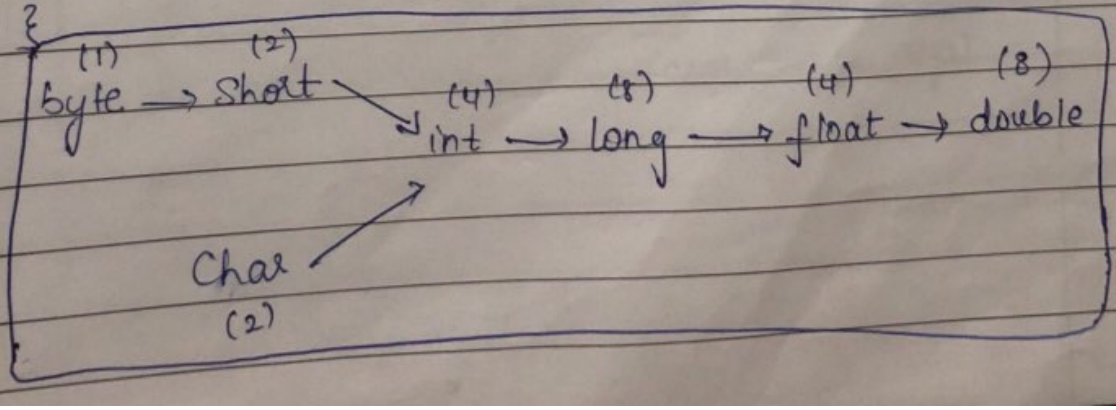
CASE-3:

```
Class A
{
  // It is not an empty class, it has Default constructor.
}
```

CASE-4:

```
Class A
{
  A()
  {
    ≡
  }
  A(int arg)
  {
    ≡
  }
  A(double arg)
  {
    ≡
  }
  A(int arg1, double arg2)
  {
    ≡
  }
}
```

```
new A();
new A(10);
new A(8.14);
new A(10,10.11)
new A('A');
new A(3.14f);
new A(100l);
new A(10,20);
```



CASE-5:

```
Class A
{
   A(int arg1, double arg2)
   {
   ==
   }
   A(double arg1, int arg2)
   {
   ==
   }
}
```

```
new A (10, 10.11);
New A (20.11, 20);
new A (10,20); //Ambiguity
new A ('A', 'B'); //Ambiguity
```

CASE-6:

```
Class A
{
   A ( int arg1, double arg2)
   {
   ==
   }
   A ( int arg2, double arg1)
   {
   ==
   }
}
```

```
new A (10, 10.11);
//Ambiguity.
```

__/__/____

METHOD OVERLOADING:

If we want to perform one task in multiple way then we will go for overloading. If we want to change the task itself then we will go for Overriding.

Definition.

1. Writing multiple methods with the same name but different in either length of arguments (or) type of arguments (or) order of occurance of arguments is known as method overloading.
2. Both static and non-static methods can be over-loaded.
3. Method Overloading is an example for Compile time polymorphism.

Overloading Static method:

```
class A
{
    public static void m1()
    {
    }
    public static void m1(int arg)
    {
    }
    public static void m1(double arg)
    {
    }
    public static void m1(int arg1, double arg2)
    {
    }
```

Overload m1() method.

```
A.m1();
A.m1(10);
A.m1(8.14);
A.m1(10,10.11);
A.m1('A');
A.m1(3.14f);
A.m1(100l);
A.m1(10,20);
```

Overloading Non Static method:

```
Class A
{
  public void m1()
  {
  ≡
  }
  public void m1(int arg1)
  {
  ≡
  }
  public void m1(double arg)
  {
  ≡
  }
  public void m1(int arg1, double arg2)
  {
  ≡
  }
}
```

Overloaded m1() Method

```
A a = new A();
a.m1();
a.m1(10);
a.m1(10.5);
a.m1(10, 10.5);
```

String[] s = new String[5];

s[0]
s[1]
.
.
s[4]

} null

30/07/2015
Monday.

Overloading Main Method.

```
class Test
{
  //overloading main method
  public static void main()
  {
    S.o.p("Running no-arg method");
  }
  public static void main(int arg)
  {
    S.o.p("Running int-arg method");
  }
  public static void main(double arg)
  {
    S.o.p("Running double-arg method");
  }
  Public static void main(int arg1, double arg2)
  {
    S.o.p("Running int/double-arg method");
  }
  public static void main(String[] args)
  {
    S.o.p("Running String[] args");
    for(int i=0; i<args.length; i++)
    {
      S.o.p(args[i]);
    }
  }
}
class Program1
{
  Public static void main(String[] args)
  {
```

Test
~~Main~~.class
Program1.class

Java program1
Java Test Hi Hello
0 1

```
        Test.main();
        Test.main(10);
        Test.main(10.14);
        Test.main(10,10.14);
        Test.
        String[] S = {"Hi", "Hello", "JSP", "Banasewadi"};
        Test.main(S); // passing array to method.
        System.out.println("Main method ended");
    }
}
```

Output:

```
java Test Hi Hello How are you.
```

Program2.

1.

```
class Calculator
{
    public void static void addNum(int Num1, int Num2)
    {
        S.o.p("Sum= "+(num1+num2));
    }
}
class Program2
    {
        public static void main(String[] args)
        {
            S.o.p("main method started");
            if(args.length <2)
            {
                S.o.p("Insufficient arguments");
            }
```

```
else
  {
   int x = Integer.parseInt(args[0]);
   int y = Integer.parseInt(args[1]);
   Calculator.addNum(x,y);
  }
 S.o.p ("main method ended");
 }
}
```

Op:

```
javac Program2.java
java Program2 10 20 // Command line prompt.
```

```
public static void m1()
public static void m1(int arg)
public static void m1(double arg)
public static void m1(int arg1, double arg2)
public static void m1(double arg1, int arg2)
```

method signature

method declaration.

```
Class Test
 {
   public int m1()
    {
     ≡
     return 10;
    }

   public void m2()
    {
     ≡
    }
 }
Test t = new Test();
 t.m1(); // Ambiguity error. CTE.
```

Note:

1. Method Overloading is a process of writing multiple method with same name but different in signature.
2. Method overloading & Constructor overloading both are examples for Compile time polymorphisms

Advantages of Method overloading

1. It reduces the Complexity for end user. That is enduser no need to remember many individual methods.

Without Overloading.

```
public void isort(int[] a)
  {
   // logic to Sort int[]
  }
public void dsort (double[] d)
  {
   // logic to sort double()
  }
public void csort( Char[] ch)
  {
   // logic to Sort Char()
  }
```

With Overloading.

```
public void Sort (int[] a)
  {
    // Copy & paste of isort
  }
public void Sort (double[] d)
  {
    // Copy paste of dsort
  }
```

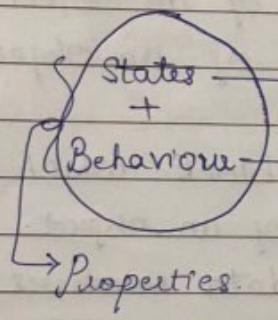
03/08/2018
Friday

```
public void Sort(Char[] ch)
{
  // Copy paste of Csort
}
S.o.println(10);
      "    ('A').
println(8.14);
```

→ Overloaded.

Array.sort( )
↓
Overloaded.

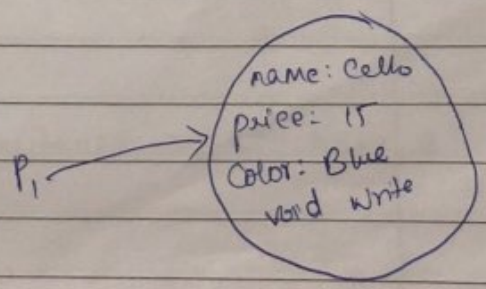
Object:



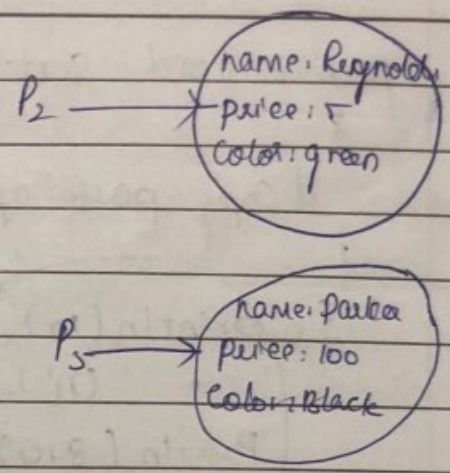
States → Characteristics of an object
Behaviour → functions of an object.
→ Properties.

```
Class Pen
{
    String name;
    int price;
    String Color;
    void write() → methods
    {
    }
}
Pen P1 = new pen();
P1.name = "Cello";
P1.price = 15;
P1.Color = "Blue";
```

Data members



```
Pen p2 = new Pen();
p2.name = "Reynolds";
p2.price = 5;
p2.color = "Green";
Pen p3 = new pen();
p3.name = "parker";
p3.price = 100;
p3.color = "Black";
```



Object:

An entity which has its own states and behaviour is known as an Object.

States represents characteristics of an object where as behaviour represents functionality of an object.

Class:

A class is a java definition block which is used to define states and behaviour of an object.

— In class states represents data members where as behaviour represents the member function.

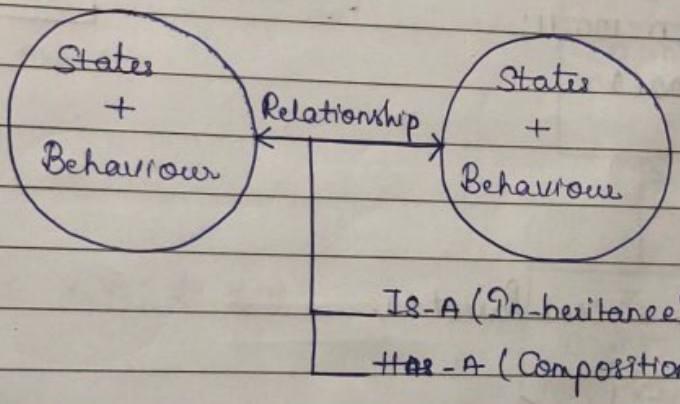
— When we create multiple objects of a class all the objects will have same states and same behaviour but different in values.

Class Diagram

04/08/18
Saturday

Student
name : String id : int marks : double
String (String, int, double) getName() : String getMarks : int SetMarks (double) : Void.

Class Diagram (UML Diagram)



Inheritance:

1. Deriving the properties from one class to another class is known as 'Inheritance'.
2. The class from where the properties are inherited is known as 'Super class'.
3. The class to which the properties are inherited is known as 'Sub-class'.
4. We can establish 'is-a' relationship b/w 2 classes by using 'extends' keyword.

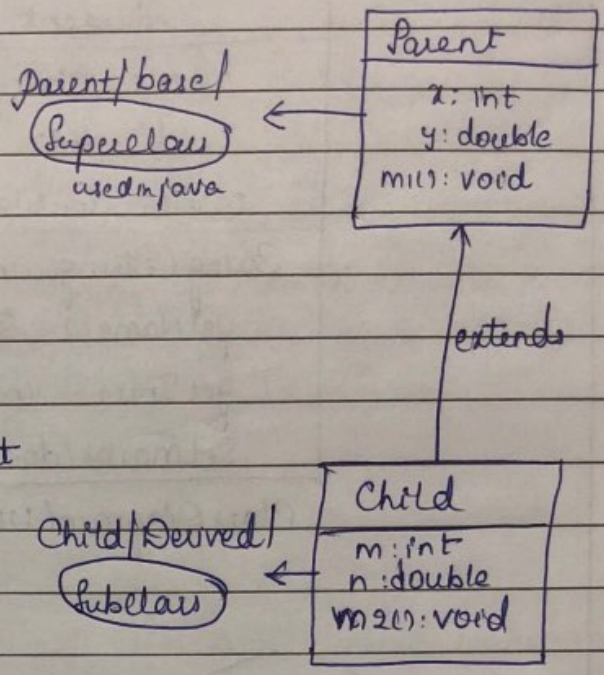
Types of inheritance:

1. Single level inheritance.

In this type of inheritance we will have a single Super class and single Sub class.

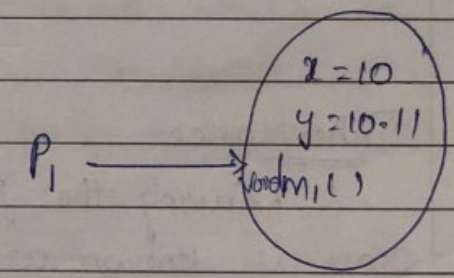
Example:

```
Class Parent
{
    int x=10;
    double y=10.11;
    void m1()
    {
    }
}
Class Child extends Parent
{
    int m=100;
    double n=100.11;
    void m2()
    {
    }
}
```



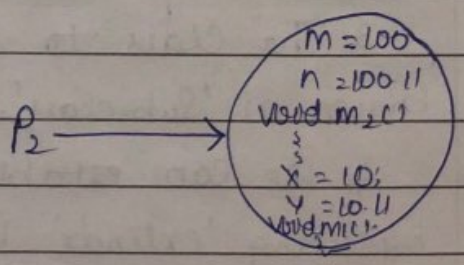

```
Parent P1 = new Parent();
    P1.x;
    P1.y;
    P1.m();
```

✓




```
Child C1 = new Child();
    C1.m;
    C1.n;
    C1.m2();
```

✓

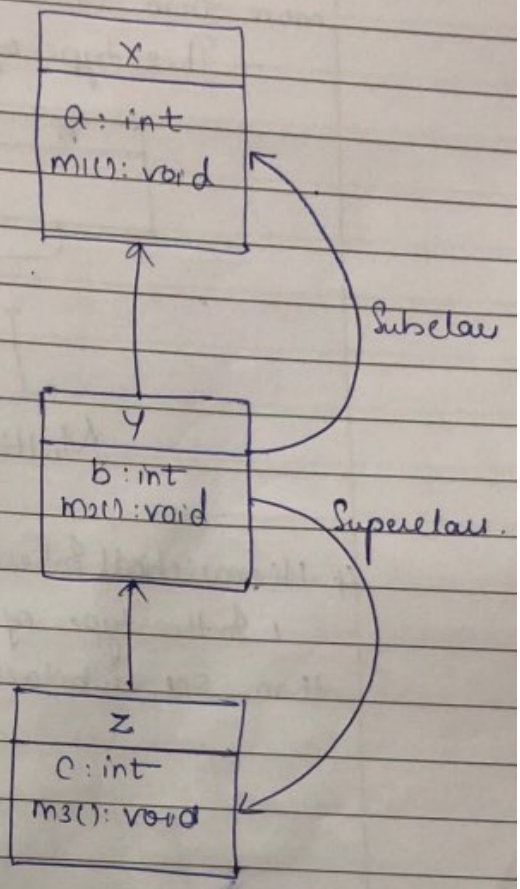



```
    C1.x;
    C1.y;
    C1.m1();
```

✓

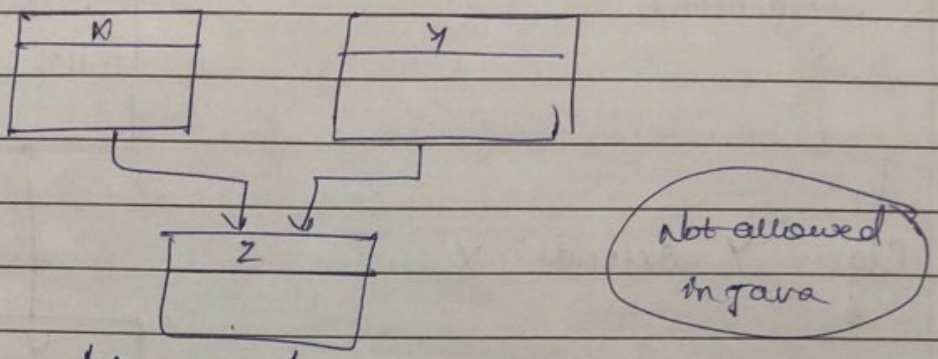
2) Multi-level Inheritance:

```
Class Parent X
{
    int a=10;
    void m1()
    {
    =
    }
}
Class Y extends X
{
    int b=11;
    void m2()
    {
    ==
    }
}
Class Z extends Y
{
    int c=12;
    void m3()
    {
    ===
    }
}
X x = new X();
    x.a;
    x.m1();
Y y = new Y();
    y.b;    y.a;
    y.m2();  y.m1();
Z z = new Z();
    z.c;
    z.m3();
    z.b;
    z.m2();
```



3) Multiple Inheritance

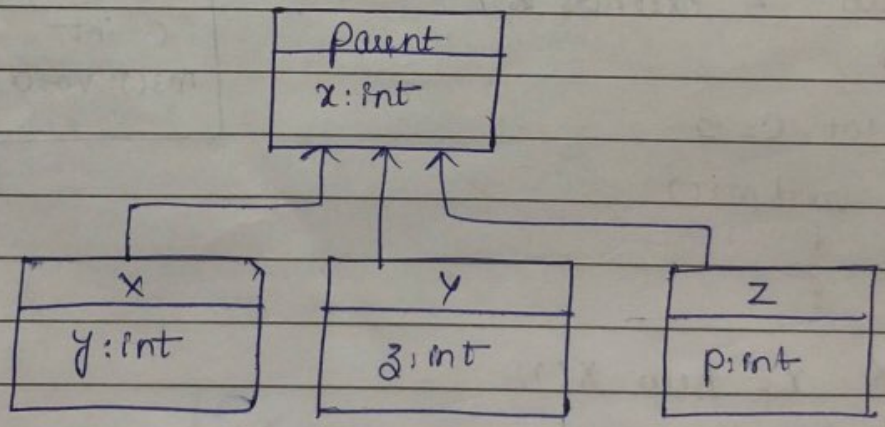
- In this type of Inheritance a sub-class will have more than one super-class.
- This type of Inheritance is not allowed in java.



Multiple-Inheritance

4. Hierarichiadl Inheritance.

1. In this type of inheritance a superclass will have more than one subclasses.



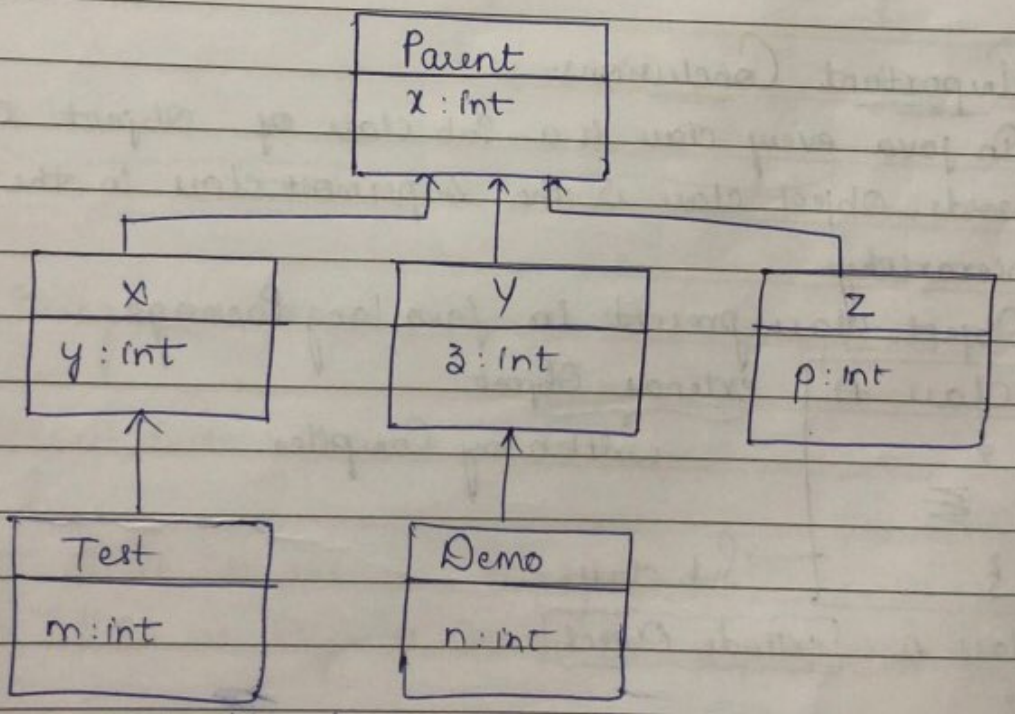
Hierarichical Inheritance

```
Class Parent
{
    int a=10;
    void m1()
    {
    }
}
```

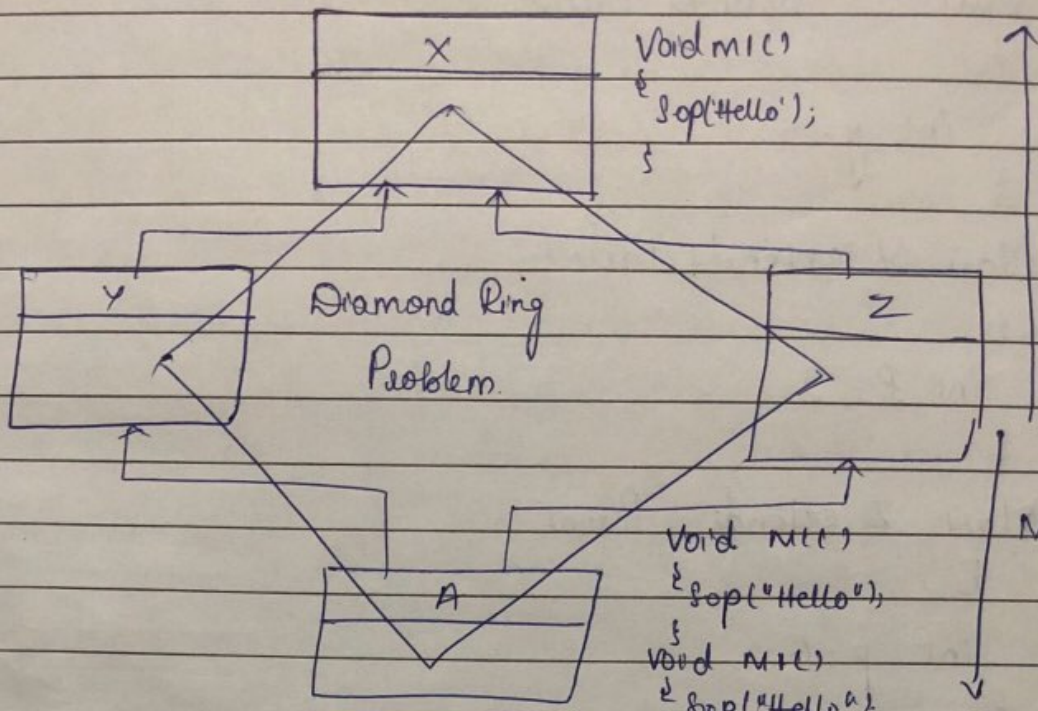
__/__/____

```
Class X extends Parent
{
    int y = 5;
}
Class Y extends Parent
{
   int B = 2;
}
Class Z extends Parent
  {
   int p = 6;
}
```

5) ~~Hybrid~~ ~~Inheritance~~



Hybrid (allowed).



Hybrid (NOT allowed) bcoz Contains multiple inheritance

Important Conclusions.

1. In java every class is a Sub-class of Object class. In other words, Object class is the supermost class in the entire java hierarichy.

2. Object class present in java.lang Package

```
Class A extends Object
{                  written by Compiler
  ≡
}  → Sub-class.
```

2
```
Class A {extends Object}
{
  =
}
Class B extends A
{
  ≡
}
Class C extends B
{
  ≡
}
```

```
class D extends C
{
}
```

3. If we declare a class as 'final' we can't inherit from such class. In other words, a final class will not have a Sub class. (It will have a Super class).

```
final class Test
{
}
```

```
class X extends Test    class Y extends Test    class Z extends Test
{                       {                       {
}                       }                       }
```

Error

Eg. String, StringBuffer, StringBuilder, wrapper classes.

Eg.
```
class Test
{
    // It is not an empty class
    // It is Default Constructor + properties of object class
}
```

4. Once we Inherit a class from another class all the properties of Super class will be inherited to Sub class except the following 2 properties.
   * private members
   * Constructors.

06/08/2018
Monday

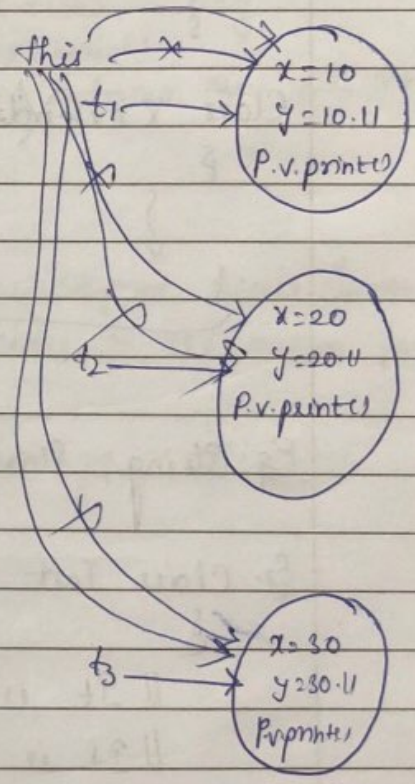
'this' Keyword:

1. It is a keyword used to refer/represent current Object.
2. An Object through which we are invoking a method is known as "Current Object".
3. This keyword will refer to only one Object at a time.
4. We need to use 'this' keyword explicitly when local variable name is same as non-Static Variablename (Instance)

Example:

```
Class Test
{
    int x;
    int y;                          → non-Static
    Test(int x, double y)
    {
        this.x = x;
        this.y = y;                 → Local
    }
    public void print()
    {
        int x=100;
        double y=100.11;            → Local
        S.o.p(x);
        S.o.p(y);
        S.o.p(this.x);
        S.o.p(this.y);
    }
}
Test t1 = new Test(10, 10.11);
Test t2 = new Test(20, 20.11);
Test t3 = new Test(30, 30.11);
t1.print(); // 10 10.11
t2.print(); // 20 20.11
t3.print(); // 30 30.11
```

(nonstatic)



__/__/____

Example:

```
Class Student
{
  String name;
  int id;
  double marks;
  Student (String name, int id, double marks)
  {
    this.name = name;
    this.id   = id,
    this.marks = marks,
  }
  public void print()
  {
    String name = "Dinga";
    int id = 420;
    double marks = 35.00;
    S.o.p("Local Student name = "+name);
    S.o.p ("Local Student id = "+id);
    S.o.p ("Local Student marks = "+marks);

    S.o.p(" Object Student name = "+this.name);
    S.o.p("Object Student name = "+this.id);
    Sop ("Object Student marks "+this.marks);
  }
}
Class Program2
  {
    Public static void main (String[] args)
    {
    Test Student S1 = new Student("Akash", 101, 80.00);
      Student S2 = new Student("Akshay", 102, 89.00);
    S1.print();
    S2.print();
  } }
```

Note:

1. 'This' keyword must be used either inside a non-static method body (or) inside a Constructor. (We cannot use inside static method).
2. All non-static can have static members but not viceversa.

Super keyword:

1. It is a keyword used to refer super-class properties from the sub-class.
2. ~~'this' key~~ 'Super' keyword must be used either inside the non-static method (or) inside a Constructor (Can't be used inside a static Method).

Example:

```
Class SuperClass
 {
    String name = "Akshay";
    int id = 10;
    double marks = 54.44;
    public void print()
      {
         S.o.p("Name" +name);
         S.o.p("Id" +id);
         S.o.p("marks" +marks);
      }
 }
Class SubClass extends Superclass
 {
    String name = "kiran";
    int id = 20;
    double marks = 75.34;
```

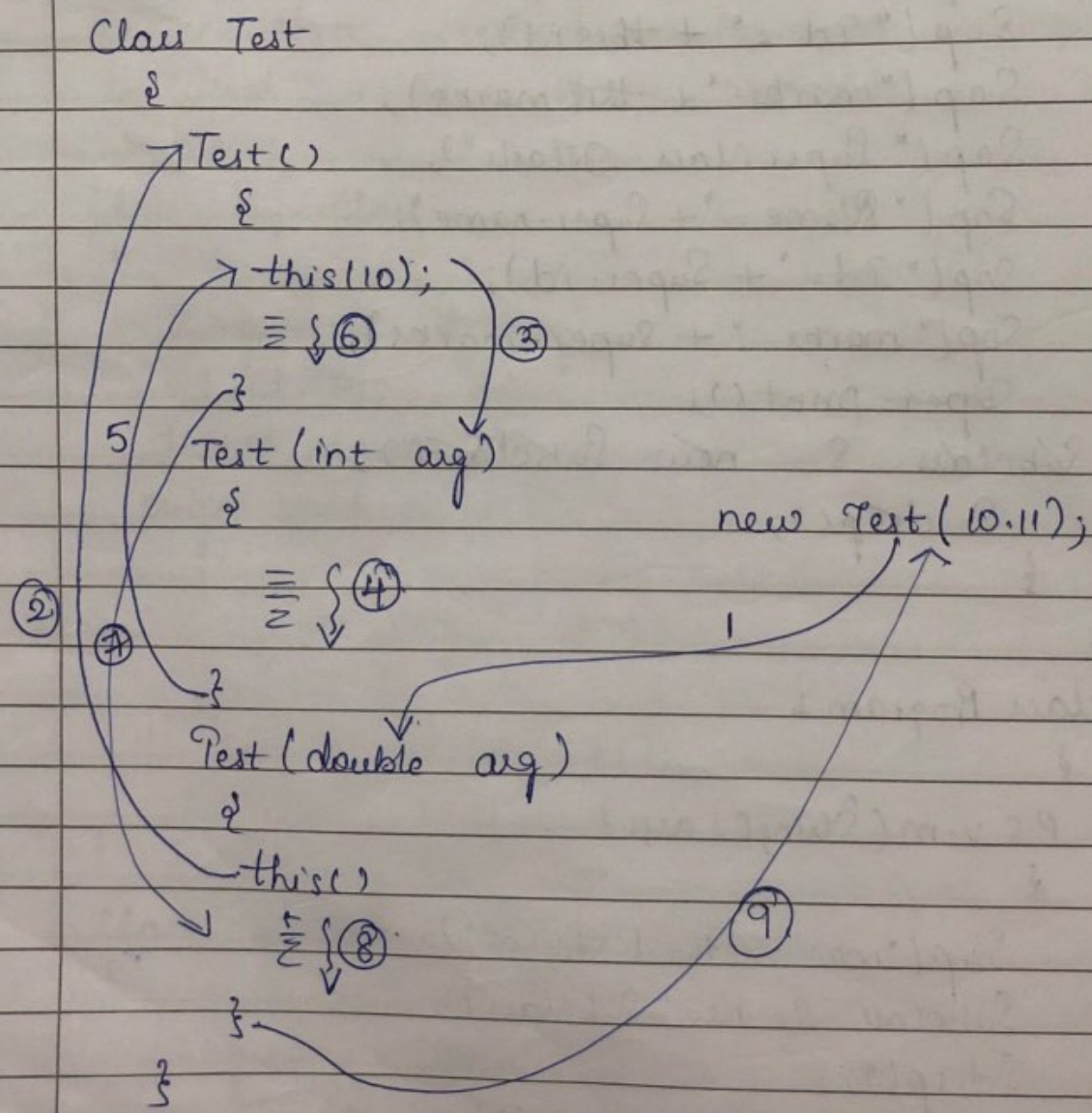
__/__/____

```
public void disp()
 {
  String name = "Omga";
  int id = 420;
  double marks = 35.00;
  S.o.p("Local Details");
  S.o.p("Name:"+name);
  S.o.p("Id = "+id);
  S.o.p("marks=" + marks);
  S.o.p("Subclass Details");
  S.o.p("Name = "+this.name);
  S.o.p("Id = "+this.id);
  S.o.p("marks="+this.marks);
  S.o.p("Superclass Details");
  Sop("Name = "+Super.name);
  Sop("Id="+Super.id);
  Sop("marks="+Super.marks);
  Super.print();
Subclass S = new Subclass();
  S.disp();
 }
}
class Program3
 {
  P.S.v.m(String[] args)
  {
   S.o.p("main method started");
   Subclass S= new Subclass();
   S.disp();
   S.o.p("main method ended");
  }
}
```

Constructor Calling

1. Calling one constructor from another constructor of the same class is known as "Constructor calling".
2. Constructor calling stmt must be the first stmt inside the constructor body.
3. Recursive Constructor invocation is not allowed. i.e. Once we call one constructor for one object we can't call the same constructor for the same object.
4.

Example Program.



We cannot use more than one Constructor call in the Constructor.

Advantage:

1. It helps us to invoke all the constructors by creating a single object.

__/__/____

Example:

1.

```
Class Test
{
  Test()
  {
  this(10,20.32);
  Sop("Running no-arg Constructor"); //3
  }
  Test(int arg)
  {
  //this(); error bcoz recursive Constructor invocation is not allowed
  S.op("Running int-arg constructor"); //1
  }
  Test(double arg)
  {
  this();
  Sop("Running double-arg Constructor"); //4
  }
  Test(int arg1, double arg2)
  {
  this(32);
  Sop("Running int/double arg Constructor"); //2
  }
}
Class Program4
{
  public static void main(String[] args)
  {
  new(10.11);
  }
}
```

07/08/2018
Tuesday

Constructor Chaining:

1. It is a process of calling Superclass Constructor from the Subclass, that class Constructor calling its Super class is known as "Constructor Chaining".
2. Constructor Chaining can be done either Implicitly (or) explicitly.
3. Implicit Constructor chaining takes place when the Super Class has no argument constructor. where as explicit Constructor chaining is required when the Super class has argument Constructor.

Advantage:

1. Constructor chaining helps us to initialize the data members of the Super class.

Example:

```
class A
{
  A()
  {
   this(434);
   S.o.p("In A class no.arg Constructor");
  }
  A(int arg)
   {
   S.o.p("In A class int-arg constructor");
   }
}
Class B extends A
 {
   B(int arg1, double arg2)
   {
    S.o.p("In B-class Constructor");
   }
}
```

__/__/____

```
Class C extends B
{
  C()
  {
  super (10, 10.11); //mandatory
  S.o.p ("In C-class Constructor");
  }
}
Class D extends C
 {
    D()
    {
    S.o.p("In D-class Constructor");
    }
}
Class Program5
 {
   P.S.V.m (String[] args)
    {
      New D();
    }
 }
```

Program-6:

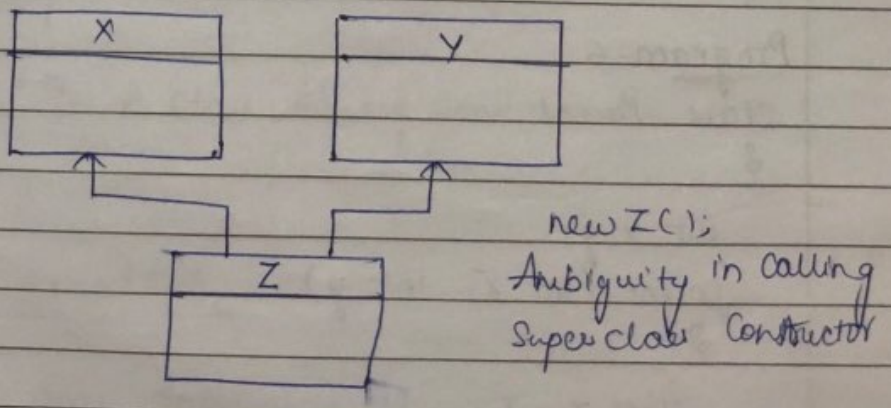
```
Class Parent
{
  int x, y;
  parent (int x, int y)
   {
     this.x = x;
     this.y = y;
   }
  public void addTwoNum()
  { Sop ("Sum = "+(x+y));
  }
}
```

```
class Child extends Parent
{
  Child(int x, int y)
  {
    super(x, y); //mandatory
  }
}

class Program6 {
    public static void main (String[] args)
    {
      Child c = new Child();
      c.addTwoNum();
    }
}
```

*** Why Multiple inheritance is not allowed in Java?

In C++;

```
class Z : public x, public Y
{
  Z().
  {
  }
}
```



1. In multiple inheritance, a Sub class will have more than one Super classes. When we create an object of a Sub class either directly or indirectly Constructor of the Super class has to be called in order to complete

08/08/2018
Wednesday.

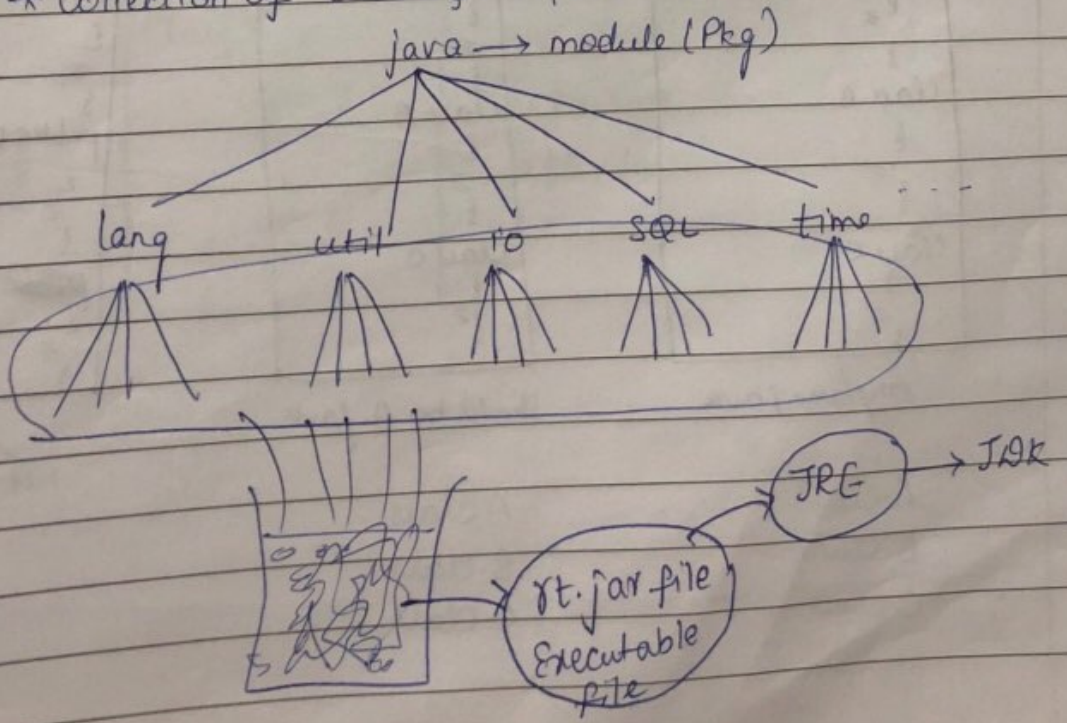
the Constructor Chaining process. Since we have more than one super classes there is an ambiguity in calling Super class Constructor. Because of this ambiguity the Constructor Chaining will remain incomplete. Hence multiple inheritance is not allowed in Java.

this, Super	this(), Super()
1. this used to refer Current object Super used to refer Super class properties from the subclass.	1. this() is used in Constructor Calling. Super() is used in Constructor Chaining.
2. this, Super can be used either inside a Constructor or non-static method.	2. These should be used inside the constructor only.
3. They can be used anywhere inside a Constructor or non-Static Method.	3. Should be used only in the first stmt inside the Constructor body.
4. They can be used together	4. Can't be used together

Packages:

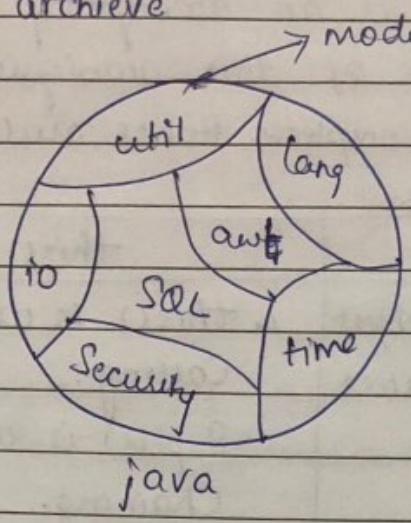
Java is Opensource

* Collection of classes & interfaces

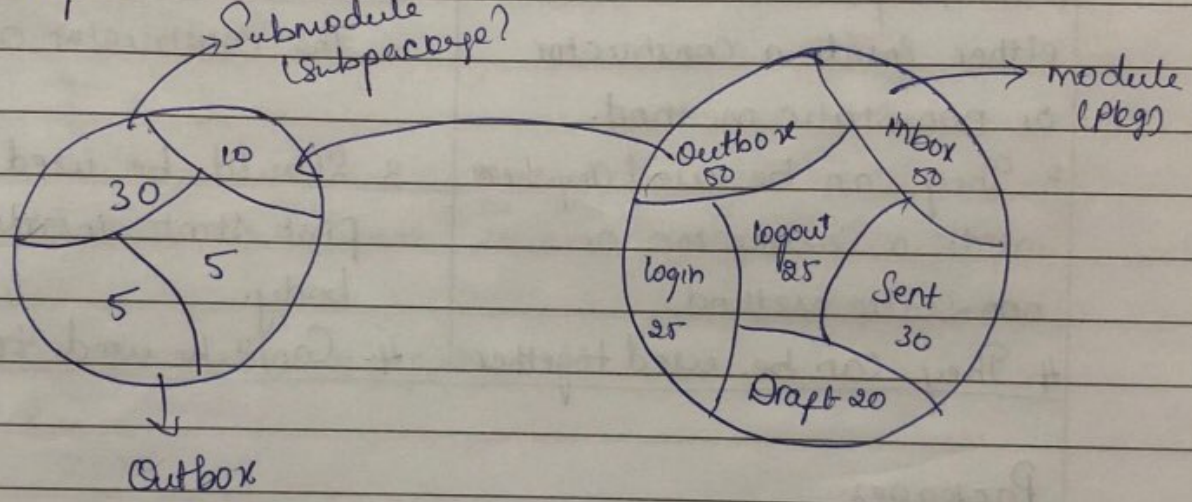


rt - run-time → Jre → JDk.

jar - java archieve



Example.




```
Class A
{
}
Class B
{
}
Class C
{
}
```

AnyName.java

```
public class A
{
}
Class B
{
}
Class C
{
}
```

Should be A.java

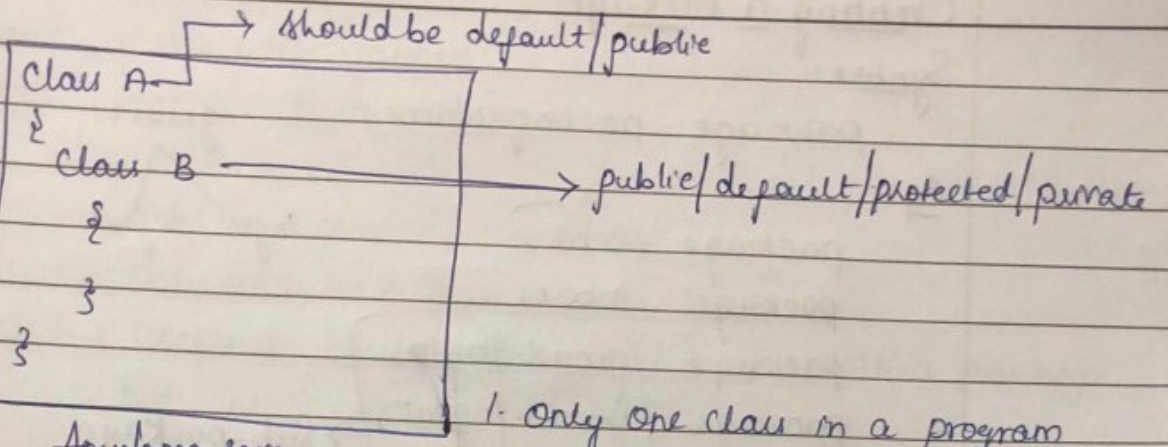
```
public class A
{
}
public class B
{
}
public class C
{
}
```

not valid / Invalid

A.class
B.class
C.class

A.class
B.class
C.class

__/__/____

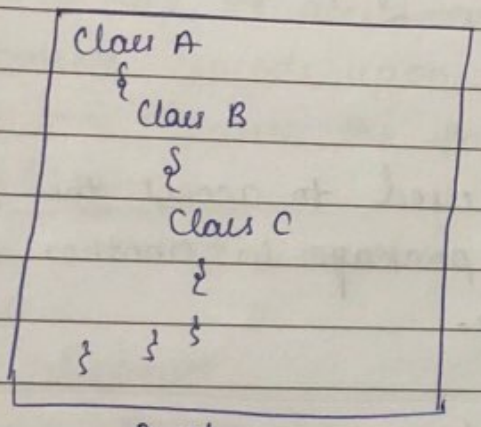


1. Only one class in a program should be public.

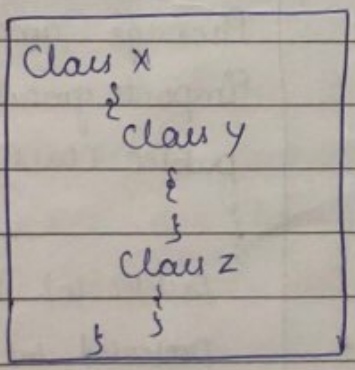
AnyName.java

A.class

A$B.class → Inner class



A.class
A$B.class
ABC.class.



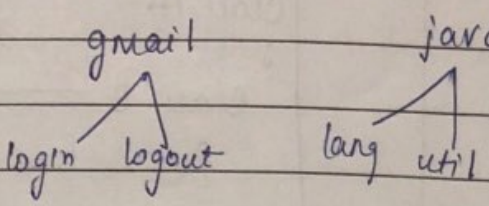
X.class
X$Y.class
X$Z.class.

09/08/2018
Thursday.

Creating a Package:

Syntax:

```
package packageName;
```



Eg:

```
package Sent;
package inbox;
package gmail.login;
package gmail.logout;
package java.lang;
package java.util;
```

→ Sub-package

↳ Parent package / super package

Import keyword:

It is a keyword used to access the properties of one class belongs to one package in another class belongs to another package.

Imports can be of 2 types.

1. Static import
2. Non-Static import.

```
package gmail.login;
import gmail.logout.Demo;
public class Test
{
    public int x=10;
    protected int y=20;
    int z=30;
    private int p=40;
    accessmodifiers
}
```

Test.java

```
Package gmail.logout;
Import gmail.login.Test;
public class Demo
{
    public int m=100;
    protected int n=200;
    int o=300;
    private int r=400;
}
```

Demo.java

import

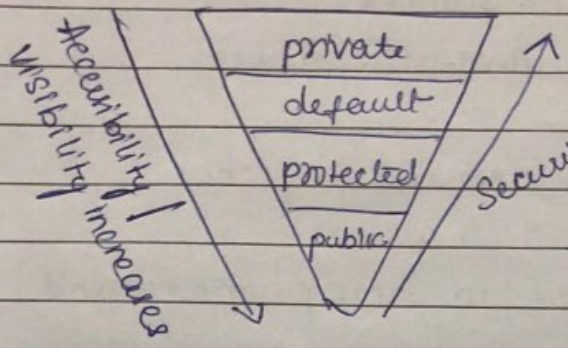
```
import java.util.Scanner;
import static java.util.Scanner;
```

Public → Can be accessed anywhere.
private → Only with in a class
protected → accessed in same package and another package but is-a relationship is necessary.
default → with in a package.

1. Just because we use import it doesn't mean we will be able to access everything. What properties are accessible depends upon access modifiers.
2. Java supports the following 4 access modifiers
   a) private:
      private members are accessable with in the class.
   b) Default:
      It is also known as package level access modifier. Default members are accessible from all the classes of the same package.
   c) Protected:
      It is same as default + protected members are also accessible outside the package provided the class where we are trying to access is having "is-a" relationship with the class to which that member belongs to.

   d) Public
      public members are accessible with in the package and also outside the package.

10/08/2018
Friday.




```
Package Pk1;
Class A
{
  public int x = 10;
}
```

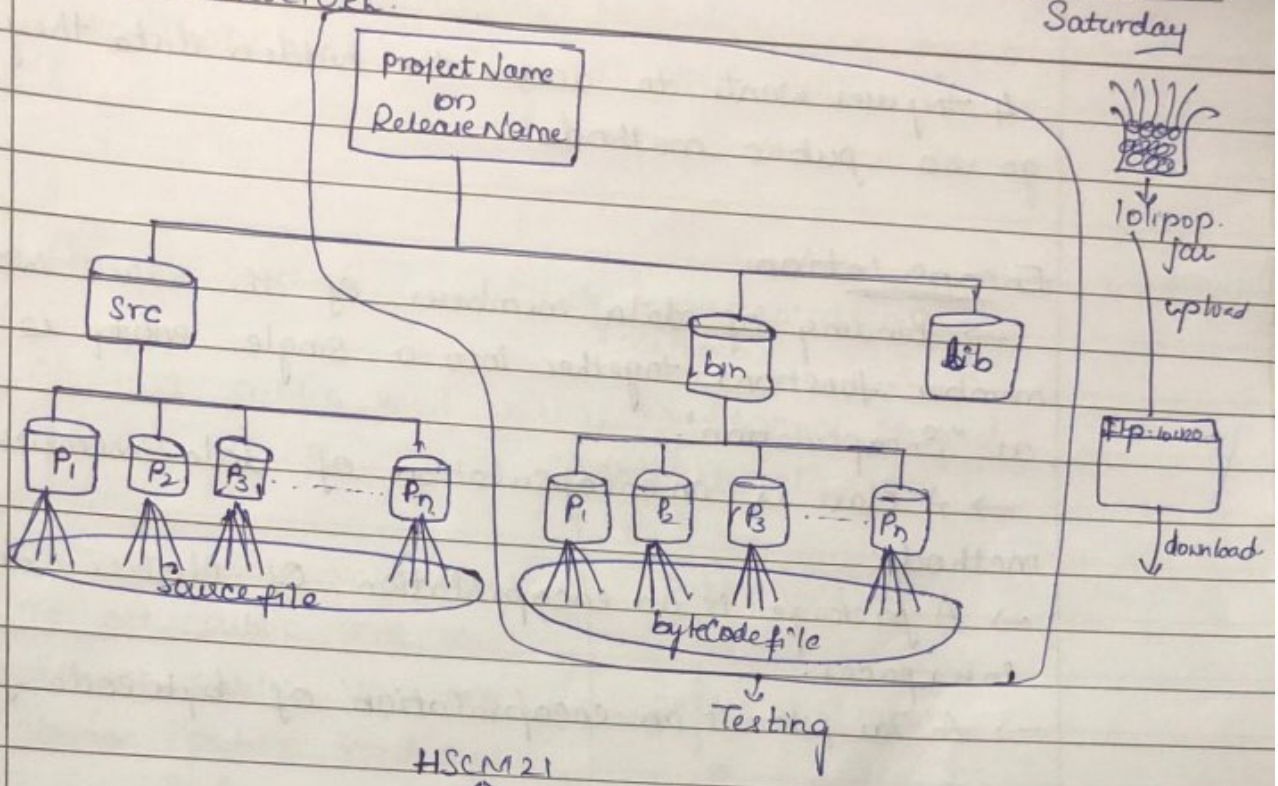
```
package Pk2;
import Pk1.A;
Class B
{
}
```

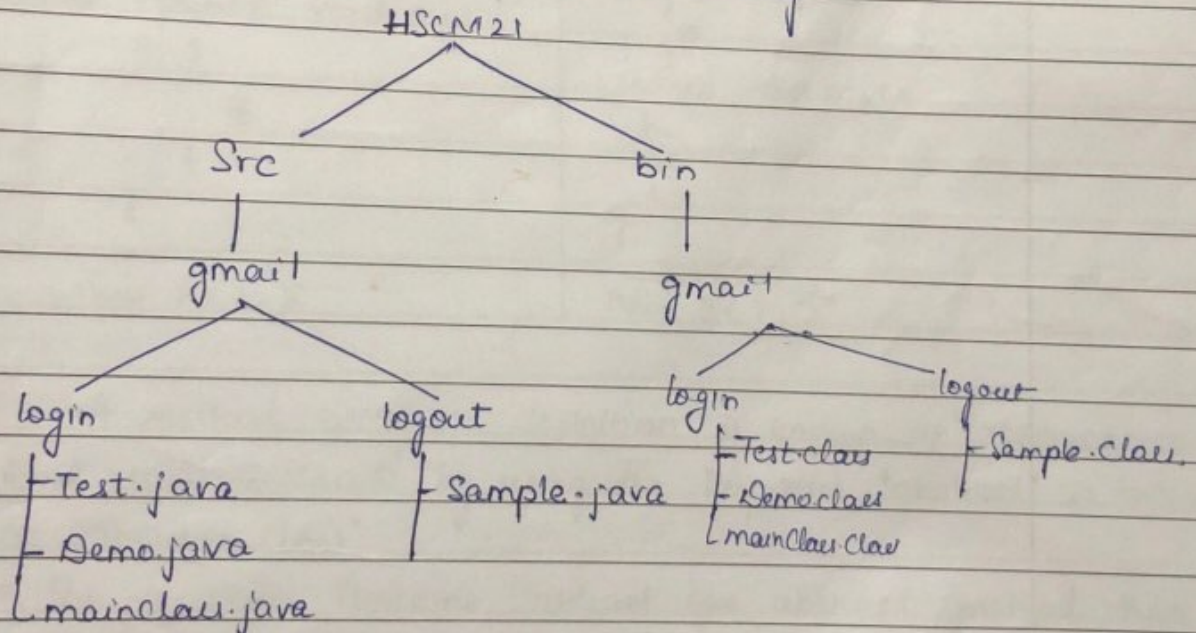
X → error

" Not allowed bcoz Class A is not public"

PROJECT STRUCTURE:

11/08/2018
Saturday





* What is data hiding?

1. Hiding the data members of a class from outside the class is known as data hiding.

2. Data hiding can be achieved by using private access modifier.

3. In all object oriented programming languages, our first priority will be on the data rather than methods. i.e. we will not allow the data to freely move around the system, Hence we try to hide the data from the outside world.

__/__/____

4. Any user wants to access the hidden data they have to go via public methods.

Encapsulation:

→ Binding of data members of the class with the member functions together into a single entity is called as "Encapsulation".

→ A class is an encapsulation of data members and methods.

→ A package is an encapsulation of classes and interfaces.

→ A Jar file is an encapsulation of byte code file.

12/08/2018

Abstract Class and Abstract Methods:

1) public void m1()
{
≡
}
Concrete Methods

2) public void m1()
{
}
Empty implementation methods.

3) abstract public void m1(); → Abstract Method.

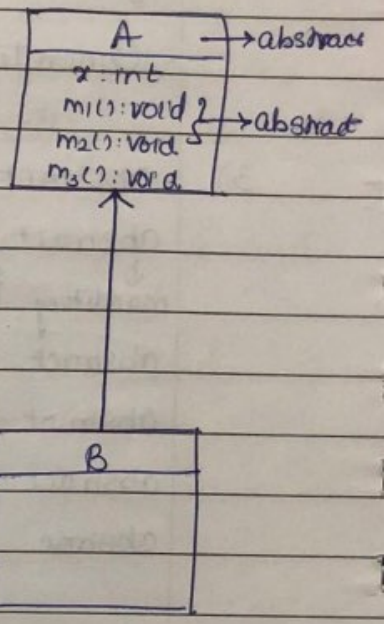
abstract class A
↓
manditory {
abstract public void m1();
abstract public void m2();
abstract public void m3();
abstract public void m4();
{
≡
}
}
new A(); ✗

abstract class B
→ optional.
{
public void m1()
{
≡
}
public void m2()
{
≡
}
public void m3()
{
≡
}
}
new B(); ✓

1. A method without definition is known as "abstract method".
2. A class declared by using the keyword "abstract" is known as "abstract class".
3. If a class contains atleast one abstract method then we must compulsorily declare the class as 'abstract'.
4. If a class contains only Concrete methods then it is optional to declare the class as 'abstract'.
5. Abstract class cannot be instantiated.
6. For all the abstract methods of abstract class, we provide the implementation in the Sub-class.
7. If Sub-class fails to Provide implementation for all the abstract methods, then declare Subclass also as abstract.

A.java

```
package ban.abstracts;
abstract public class A
{
  public int x=100;
  abstract public void m1();
  abstract public void m2();
  public void m3()
   {
    S.o.p("Running Concrete method");
   }
}
```



B.java

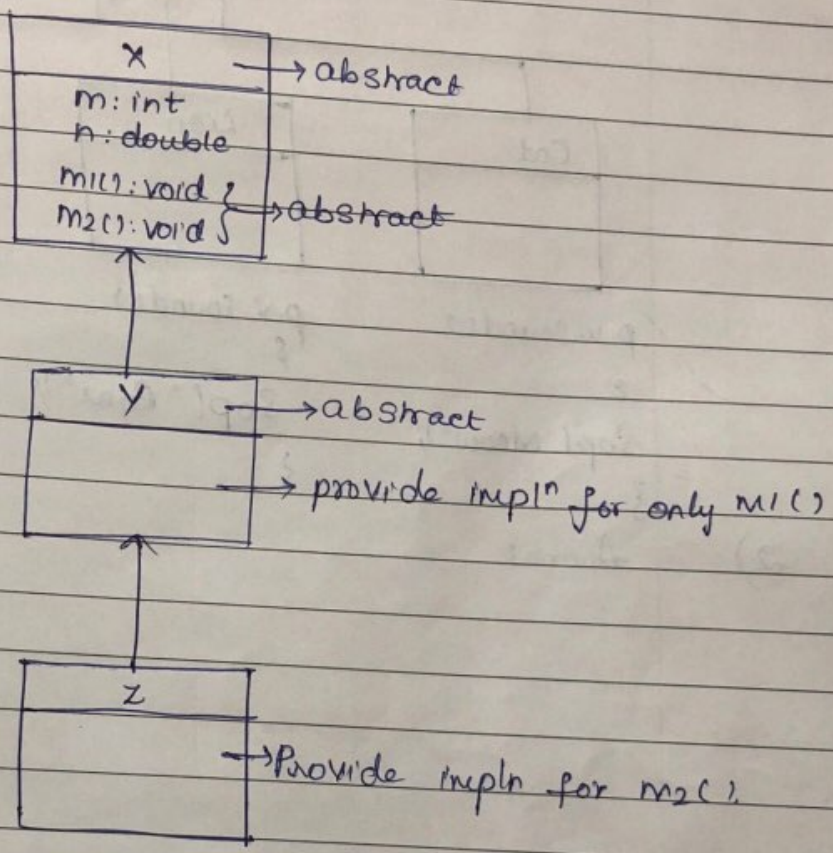
```
package ban.abstracts;
public class B extends A
 {
  public void m1()
   {
    S.o.p("first implementation method");
   }
  public void m2()
   {
    S.o.p("Second implementation method");
   }
}
```

MainClass.java:

```
package ban.abstracts;
public class MainClass
{
  public static void main(String[] args)
   {
    S.o.p("main method Started");
    B b = new B();
    b.m1();
    b.m2();
    b.m3();
```

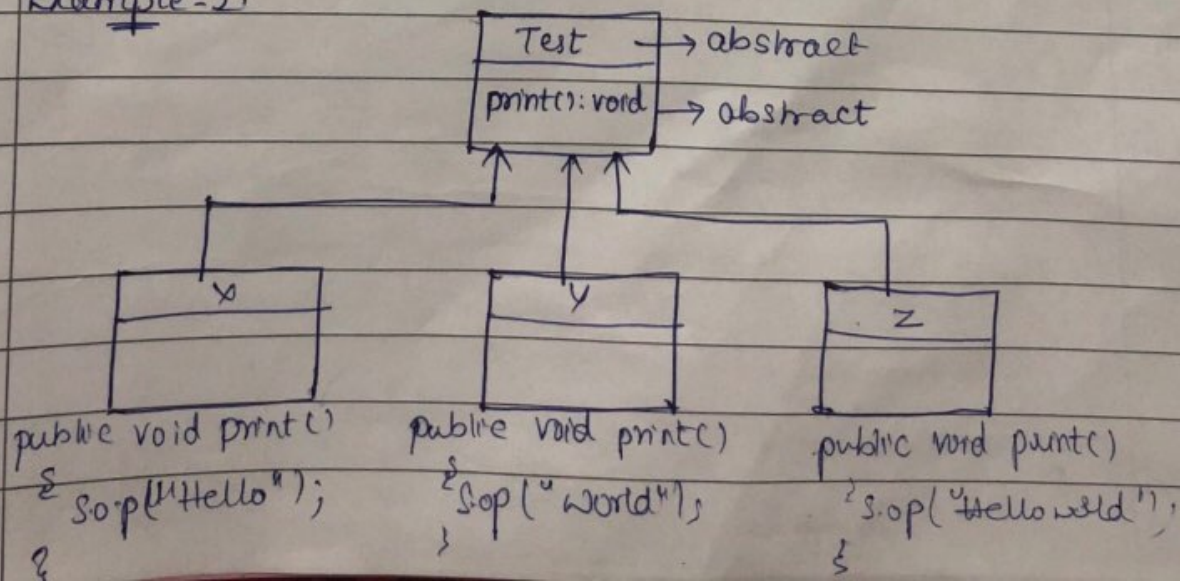
S.o.p ("x = " + b.x);
S.o.p ("Main method ended");
}
}

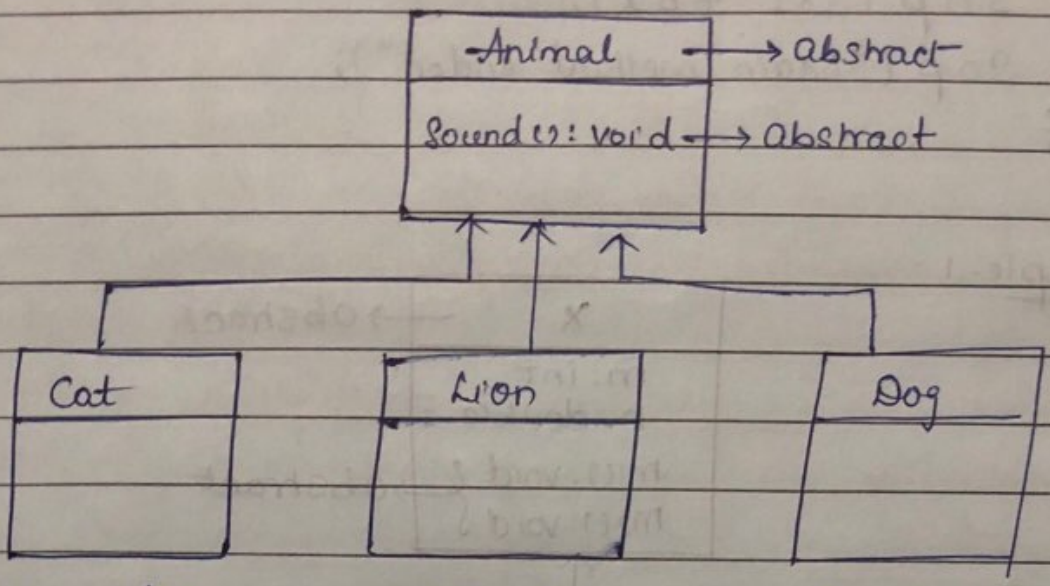
Example-1



Z z = new Z();
z.m1();
z.m2();
S.op(z.m);
S.op(z.n);

Example-2.






```
p.v. Sound()          p.v. Sound()          p.v. Sound()
{                     {                     {
Sop("Meow");          Sop("Roar");          Sop("Bow");
}                     }                     }
```

2) Animal

13/08/2018
Monday
Tuesday

Important Conclusions:

1. Case-1:

Static Abstract class can contain static members also which can be accessible by using the class name.

Example:

```
abstract class A
{
    public static int x=10;
           int y=20;
    public static void m1()
    {
    ≡
    }
    public static void m2()
    {
    ≡
    }
    abstract public void m3();
}
S.o.p(A.x);
So. A.m1();
```

2. → While providing the implementation in the sub-class either it should retain the same access modifier (or) increase the visibility but cannot be decreased.

→ Since private methods don't inherit hence they can't be declared as abstract.

Example:

```
class abstract class A
    {
        abstract public void m1();
        Abstract void m2();
    }
    class B extends A
    { void m1()
    public void m2() { — }
```

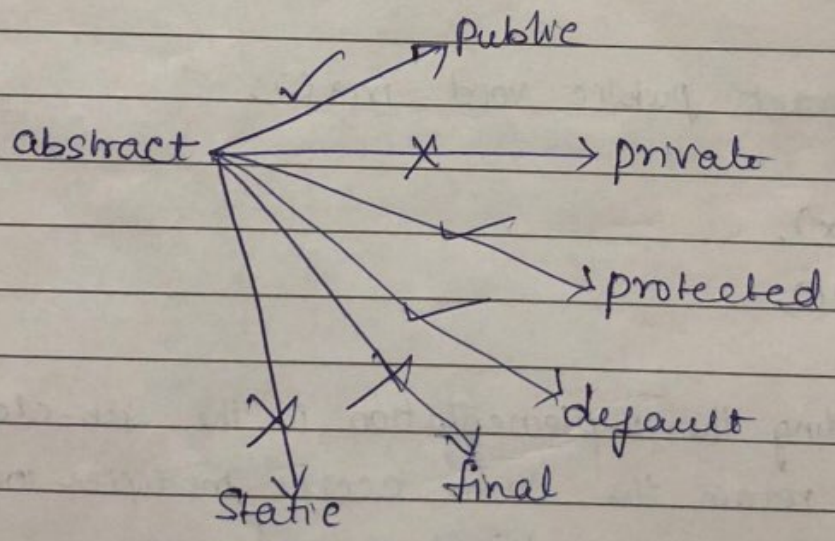
✓ ✗ error

abstract class ⇒ public ✓ protected ✓ default ✓ ~~private~~

Subclass ⇒	public ✓	protected ✓	default ✓
	private ✗	public ✓	protected ✓
	default ✗	default ✗	public ✓
	protected ✗	Private ✗	private ✗

3. Abstract class cannot be declared as 'final', because final class doesn't have subclasses.

4. Abstract and static will never exists together. i.e., abstract method cannot be static and viceversa.



5. → Though abstract class cannot be instantiated, abstract class allows constructors.

→ Abstract class constructor will be executed when we create an object of sub-class through constructor chaining process.

→ Constructor in the abstract class are used to initialize the data members of abstract class

Calculator.java

```
package ban.abstracts;
abstract public class Calculator
{
   int x, y;
   public Calculator(int x, int y)
   {
     this.x = x;
     this.y = y;
   }
   abstract public void adder();
}
```

Addition.java

```
package ban.abstracts;
public class Addition extends Calculator
{
   public Addition(int x, int y)
    {
      Super(x, y);
    }
   public void adder()
    {
      S.o.p("Sum=" + (x+y));
    }
}
```

MainClass.java

```
package ban.abstracts; import java.util.Scanner;
public class MainClass
  {
     P.S.V.m(String[] args)
       {
          S.o.p("Main method started");
```

16/08/2018
Wednesday

```
Scanner sc = new Scanner(System.in);
S.o.p("Enter the first number");
int num1 = sc.nextInt();
S.o.p("Enter the second number");
int num2 = sc.nextInt();
Addition a = new Addition(num1, num2);
s.add();
System.out.println("Main method ended");
    }
}
```

Interfaces:

Advantages of abstract class:

1. Abstract class helps us to enhance the application.

When to go for abstract class:

1. When we know the partial implementation of an object then we will go for abstract class.

16/08/18
~~Wednesday~~ Thursday

Interfaces:

1. It is a java definition block used to define the data members and methods.
2. Data members are by default static and final where as methods are by default public and abstract.

Syntax:

```
interface interfaceName
{
  // by default Data members are static and final.

  // by default methods are public and abstract.
}
```

```
interface Itr1
{
static final int x=10;
abstract public void m1();
}
```

Same ⟺

```
interface Itr2
{
int x=10;
void m1();
}
```

⟶ extends
⟶ implements

3. For all the abstract methods of an interface, we provide the implementation in the implementation class by using the keyword 'implements'.
4. If implementation class failed to provide implementation for all the abstract methods then declare implementation class as abstract.
5. Just like abstract class even interface cannot be instantiated.

Notes

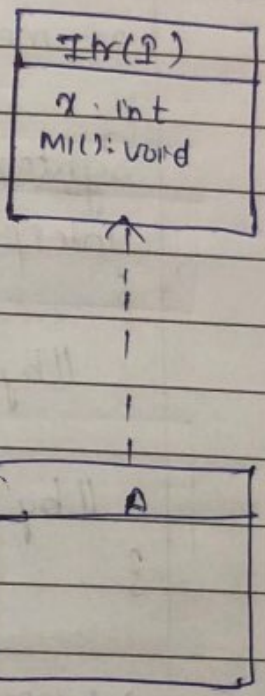
Class extends Class
Interface extends Interface
Class implements Interface

__/__/____

Program:

Itr.java:

```
package ban.interfaces;
public interface Itr
{
    int x=10;
    void m1();
}
```



A.java:

```
package ban.interfaces;
public class A interface implements Itr
{
    public void m1()
    {
     S.op ("Implementation method");
    }
}
```

Implementation class

MainClass.java:

```
package ban.interfaces;
public class MainClass extends A
  { P.S.V.M()
    {
    System.out.println ("Main Method Started");
    A a = new A(); a.M1(); Sop(Itr.x);
    System.out.println ("main method ended");
    }
}
```

1. Just like a class an interface can also extend from another interface but not from class.

X.java

```
public interface X
{
    int m=10;
    void m1();
}
```

Y.java

```
public interface Y extends X
{
    public int n=80;
    void m2();
}
```

Z.java

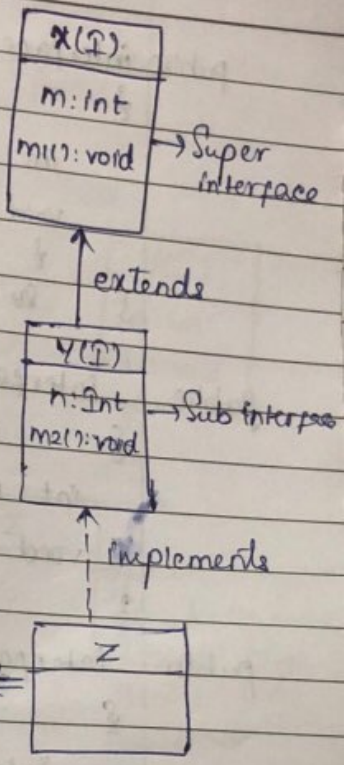
```
public class Z implements Y
{
  (public static void main(_) else declare as abstract
   {)
    Sop("main method started")
    Sop(X.m);
    Sop(Y.m);
    Sop(Y.n);
    Z z = new Z();
    z.m1();
    z.m2(); Sop("main method ended");
    }
}
```

Sop(X.m);

Sop(Y.n);
Sop(Y.m);
Z z=new Z();
z.m1();
z.m2();

Provide implⁿ for both m1() & m2()

Provid implementation for m1() & m2()

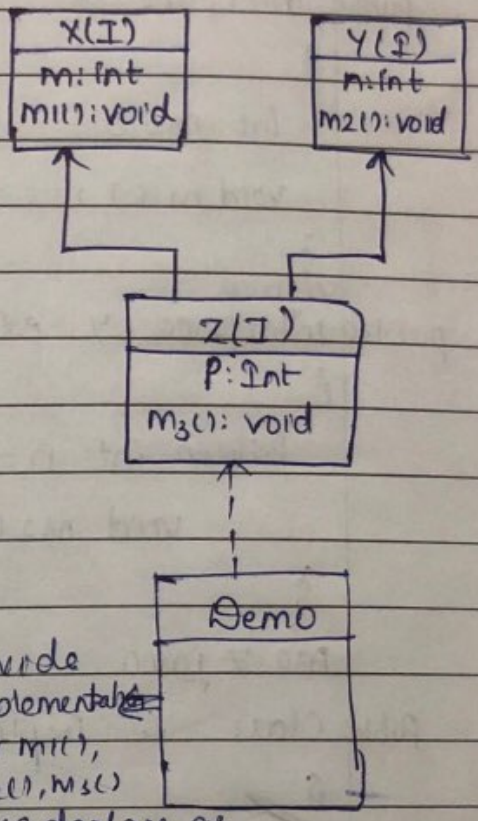


→ A class cannot extend from more than one class where as an interface can extends more from more than one interface.

→ Multiple inheritance with respect to interfaces are allowed but not with respect to classes. This is bcoz

→ Constructors are not allowed inside an interfaces. Since there is no constructor, no question of ambiguity.

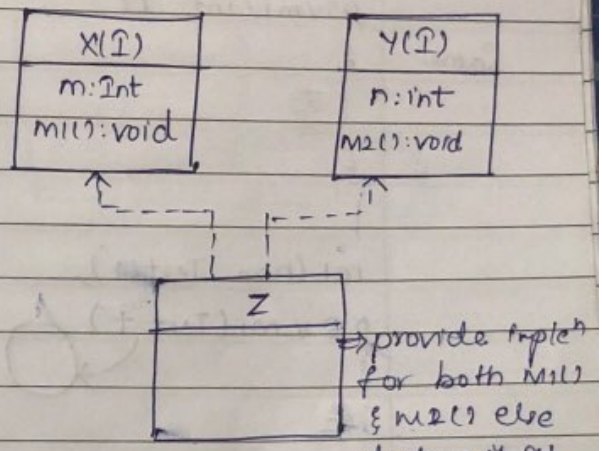
```
public interface X
{
    int m=10;
    void m1();
}

public interface Y
{
    int n=20;
    void m2();
}

public interface Z extends X,Y
{
    int P=30;
    void m3();
}

public class Demo implements Z
{
    P.S.V.m()
    {
        Sop("main method started");
        Demo d= new Demo();
        Sop(Z.m);
        Sop(Z.n);
        d.m1();
        d.m2(); Sop("main method ended");
    }
}
```



__/__/____

⇒ A class can provide implementation for more than one interface.

```
public interface X
{
  int m=10;
  void m1();
}
public interface Y
{
  int n=20;
  void m2();
}
public class Z implements X,Y
  {
     public void m1()
      {
       Sop("m1()");
      }
     public void m2()
      {
       Sop("m2()");
      }
  }
public class MainMethod
  {psv.m()
     {
      Sop("mms");
      Z z = new Z();
      S.op(z.m1());
      Sop(z.m2());
   Sop(X.m);
   Sop(Y.n); Sop("mmE");
  }
```



17/08/2018
Friday

Passing / Returning Objects from / To methods

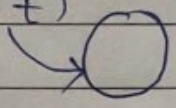
```
m1(10);
psvm1(int x)
{
  ≡
}
```

Same

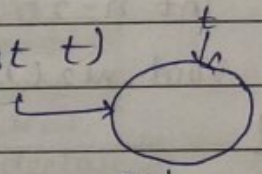
```
{ int x = 10;
  m1(x);
}
P.S.vm1(int x)
{
  ≡
}
```

different x because both are different method they are local variables

```
m1(new Test());
p.s.v m1(Test t)
{
  ≡
}
```



```
{ Test t = new Test();
  m1(t);
}
P.S.V.m1(Test t)
{
}
```



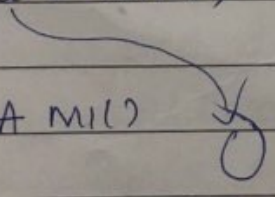
2t's are different

Returning Objects from/to method:

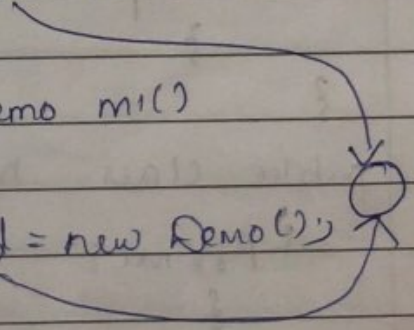
```
double d=m1();
P.S. double m1()
{
  ≡
  return 8.14;
}
```

```
Char c = m1();
P.S. Char m1()
{ Char c='A';
  ≡
  return c;
}
```

```
A a = m1();
P.S. A m1()
{
  ≡
  return new A();
}
```

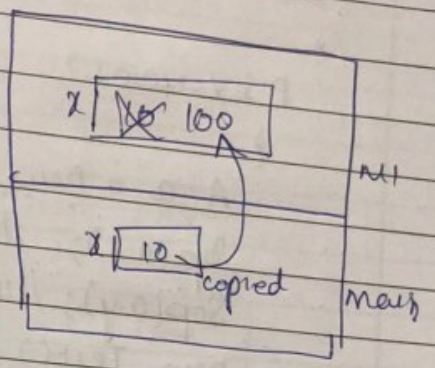


```
Demo d = m1()
P.S. Demo m1()
{
  Demo d = new Demo();
  ≡
  return d;
}
```



Eg:

```
class A
{
    P.S.V main()
    {
        int x=10;
        Sop(x); //10
        M1(x);
        Sop(x); //10
    }
}
P.S.V M1(int x)
{
    Sop(x); //10
    x=100;
    Sop(x); //100   Call by value
}
```



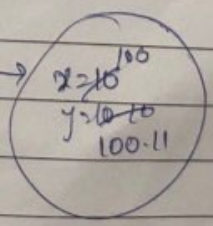
Note:

Java Supports call by value ~~address passing~~

→ Call by reference is not allowed in java. If it comes, there is pointer. In java pointer is not allowed.

Eg: 2

```
class A
{   int x=10;
    double y=10.11;
}
class Test
{
    public void modify(A a)
    {
        Sop(a.x);
        Sop(a.y);
        a.x=100;
        a.y=100.11;
    }
}
```

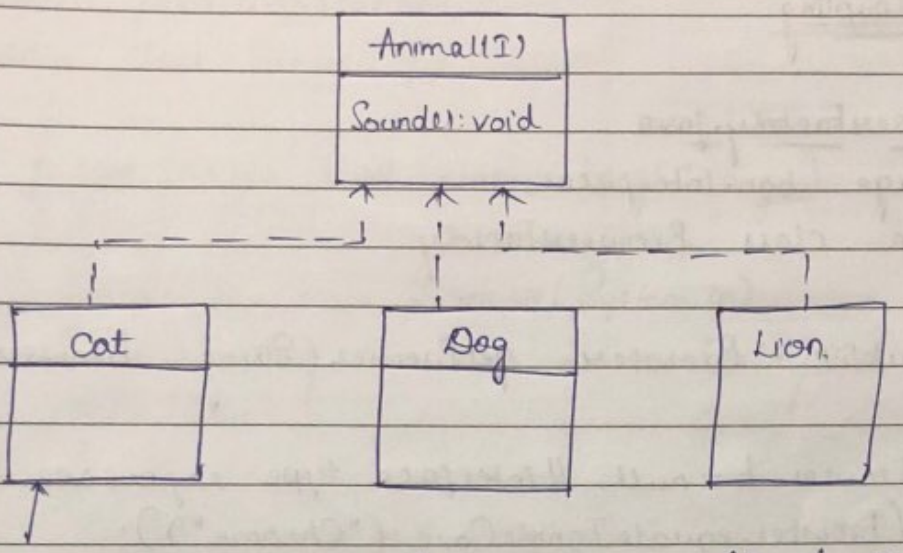


__/__/____

```
Class MainClass
{
  P.S.V. main()
  {
    A a = new A();
    Sop(a.x); //10
    Sop(a.y); //10.11
    new Test().modify(a); //Here it will be changed
    Sop(a.x); //100          bcoz Obj is not passed
    Sop(a.y); //100.11       Only reference will be passed
  }                                (Shares the location)
}
```

20/08/2018
Monday.

An interface can have any number of implementation classes.




```
Cat c= new Cat();      Dog d=new Dog();      Lion l = new Lion()
c.Sound();             d.Sound();            l.Sound();
```

Interface type reference variable → Tight Coupled program

```
Animal a = new Animal;
```

↳ Can refer to any of its implementation class object

```
Animal a = new Cat();
a.Sound(); // Cat sound
a= new Dog();
a.Sound(); // Dog sound
a= new Lion();
a.Sound(); // Lion sound
```

} Loosely Coupled program

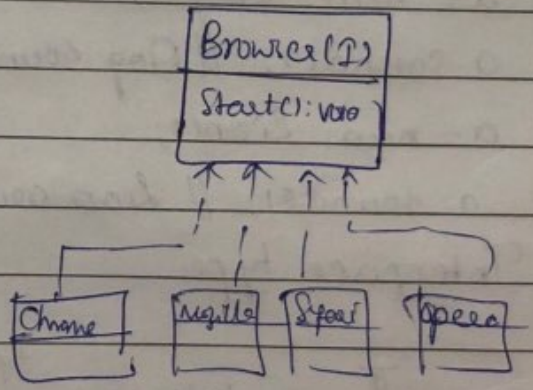
Interface type #

Though interface type cannot be instantiated, a reference variable of interface type is allowed. This interface type reference variable can refer to any of its implementation class object.

→ Interface type reference variable helps us in achieving Loose Coupling.

BrowserFactory.java

```
package ban.interfaces;
public class BrowserFactory
{
    public Browser getBrowser(String browser)
{
    Browser b=null; // interface type reference variable
  if (browser.equalsIgnoreCase("Chrome")).
    {
      b= new Chrome();
    }
else if (browser.equalsIgnoreCase("mozilla"))
    {
      b= new mozilla();
    }
else if (browser.equalsIgnorecase("Safari");
    {
      b= new safari();
    }
else
    {
      b= new Opera();
    }
    return b;
}
}
```



BrowserApp.java

```
package
import java.util.Scanner;
Public class BrowserApp
{
  public static void main(String[] args)
  {
    Scanner sc = new Scanner(System.in);
    BrowserFactory b = new BrowserFactory();
    while(true)
    {
      Sop("1.Chrome In 2.mozilla In 3.Safari In 4 Opera In 5.Exit");
      System.o.p("Enter your Choice");
      int ch = sc.nextInt();
      Switch(ch)
      {
        Case 1: b.getBrowser("Chrome").Start();
                break;
        Case 2: b.getBrowser("Mozilla").Start();
                break;
        Case 3: b.getBrowser("Safari").Start();
                break;
        Case 4: b.getBrowser("Opera").Start();
                break;
        Case 5: System.Exit(0);
      }
    }
  }
}
```

Browser.java

```
public Interface Browser
{
  public void Start()
    {
      Sop("Browser Started");
    }
}
```

Chrome.java.

```
public Class Chrome implements Browser
{
  public void Start()
    {
      S.o.p("Chrome Browser Started");
    }
}
```

Mozilla.java

```
public Class Mozilla implements Browser
  {
    public void Start()
      {
        S.op("Mozilla Browser Started");
      }
  }
```

Safari.java

```
public Class Safari implements Browser
  {
    Public void Start()
      {
        Sop("Moz.Safari Browser Started");
      }
  }
```

21/08/2018
Tuesday.

Opera.java:

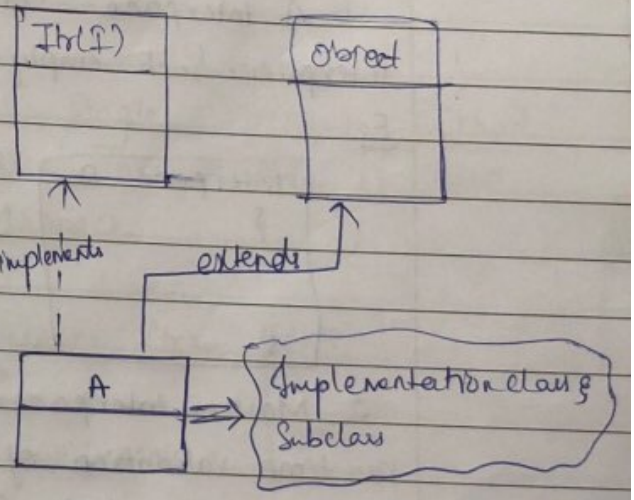
```
public class Opera implements Browser
{
    public void Start()
    {
        S.op("Opera browser Started");
    }
}
```

Example:

A class can access acts as Sub-class as well as implementation class simultaneously.

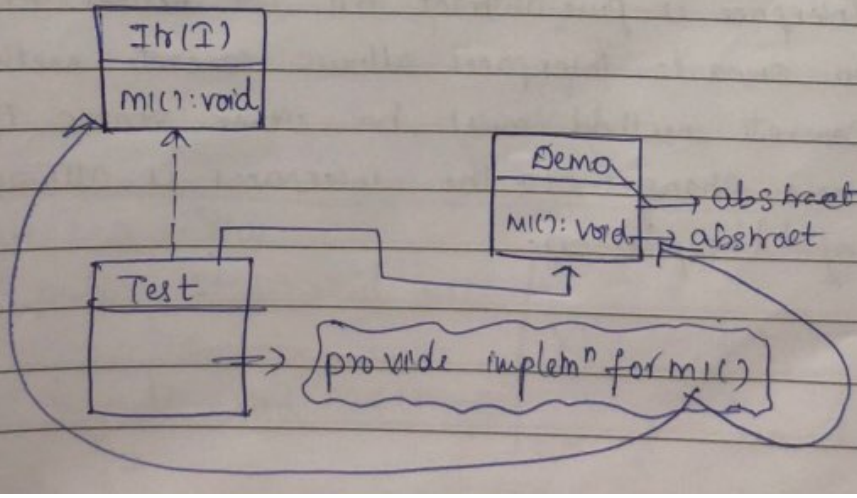
Eg:

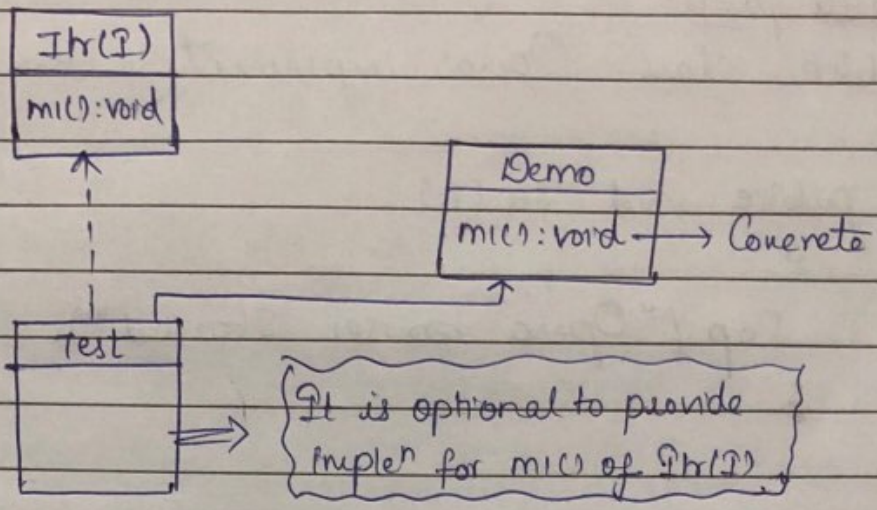
```
Interface Itr
{
    Void M1();
}
Class A extends Object implements Itr
{
    public void M1()
    {
    }
}
```



i) Class A extends Object class implements Itr ✓

ii) Class A implements Itr extends Object ✗





Important Conclusions:

Case-1:

1. A interface contains nothing is known as empty interface. Such empty interfaces are called as marker interfaces.

Eg:

```
interface A
{


}
```

Empty interfaces
Marker interfaces.
↳ JVM
↳ Run time validation

2. Marker interfaces are used by the JVM for the run time validation of an object.

Eg: Clonable (I)
Random Access (I)
Serializable (I)

Case-2:

Interface is pure abstract till 1.7 Version. From 1.8 Version onwards interfaces allows concrete methods also. This Concrete method must be either static (or) default. This new changes to the interfaces is allowed as a part of 'λ' expression.

Eg: package ban.interfaces;
public interface Itr1
{
public static void M1()
{
System.out.println("Static Concrete method");
}
default void M2()
{
S.op("non-Static Concrete method");
}
}

Case-3:

An interface which contains Single abstract method (SAM) is known as "functional interface". This feature is also introduced as a part of 1.8 changes.

```
interface Itr1
{
  void M1();
}
```
✓

```
interface Itr2
{
  Void M1();
}
```
✓

functional interface

Eg: Runnable (I)
Comparable (I)
Callable (I)

```
interface Itr3
{
  public static void M1()
  {
  }
  void M2();
}
```
✓

```
interface Itr4
{ public static void M1()
  {
  }
  default void M2()
  {
  }
}
```
✗

```
interface Itrs
{
   void M1();
   void M2();
}
```

X

Case:

4. Difference between abstract class and an interfaces.

Abstract class	Interface
1. Contain both abstract and Concrete methods.	1. Only abstract methods (till 1.7) from 1.8 Concrete is allowed
2 Constructors are allowed	2 Constructors are not allowed
3. Datamember can be static (or) non-static.	3 Datamembers by default static & final
4 A class cannot extend from more than one class.	4. An interface can be extended from more than one interface
5. Abstract Method implementation we provide in the subclass.	5. We provide in implementation class

Note:

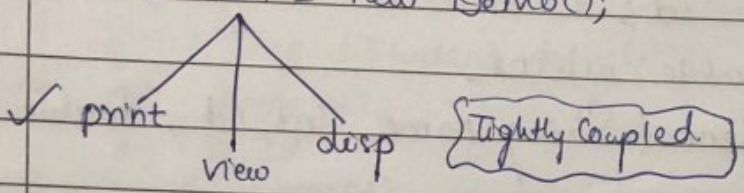
When interface type reference variable refers to an implementation class object. Using the interface type reference variable we can access only implementation methods (class specific methods are cannot access). To Access class specific methods we need to do typecasting.

Eg:

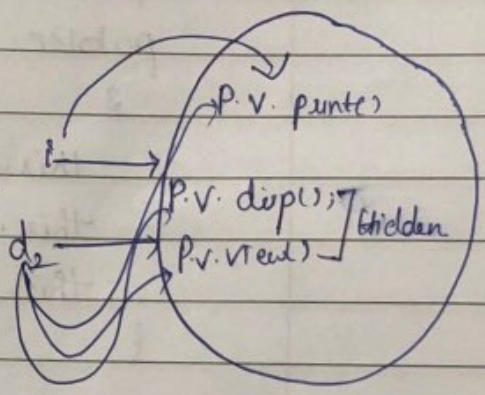
```
interface Itr
{
   void print();
}
class Demo implements Itr
{ public void print() { } implementation method
  {
  }
```

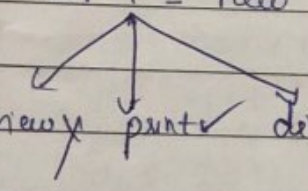
public void disp()
{
≡
}
public void view()
{
≡
}
} Class Specific Method

Demo d1 = new Demo();

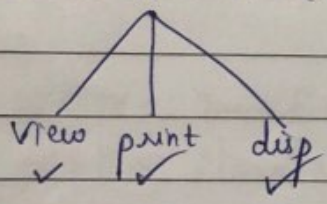


Ihr i = new Demo(); // loose Coupling.





Demo d2 = (Demo) i; // Type Casting



Java Bean class.

Writing a public class with private data members, public constructor, getters and setters known as Java Bean class.

Getters are used to get the values of private data members where as ~~setter~~ Setters are used to set the values of private data members.

Java Bean class is a very good example for data hiding and encapsulation

Java Bean class is used to achieve DAO(Data Access Object) & DTO(Data Transfer Object) layers of project development.

Eg: Student.java:

```
public class Student
 {
   Private String name;
   private int id;
   private double marks;
   public Student (String name, int id, double marks)
    {
     this.name = name;
     this.id = Id;
     this.marks = marks;
    }
public String getName()
 {
  return name;
 }
public void setName(String Name)
 {
  this.name = name;
}
public int getId() {
  return Id;
}
public void SetId (int id) {
  this.id = id;
}
public double getmarks () {
 return marks;
}public void setMarks (double marks) {
 this.marks = marks;
}
}
```

MainClass.Java

```
Class MainClass
{
 P.S.V.m(String[] args)
 {
  Student S1 = new Student ("Anga", 10, 81.41);
  Student S2 = new Student ("Angi", 20, 82.45);
  Sop ("first Student details");
  Sop ("Name :"+S1.name);
  Sop ("Id = "+S1.Id);
  Sop ("Marks="+S1.Marks);
  Sop ("Second Student details");
  Sop (S2.name);
  Sop (S2.Id");
  Sop (S2.Marks);
  S1.SetMarks (55.45);
  S2 Set marks (88.46);
  Sop ("first ");      Sop ("Second ");
  Sop(S1.name);        Sop (S2.name);
  Sop(S1.Id);          Sop(S2.id);
  Sop(S1.marks);       Sop (S2.Marks);
 }
}
```

Eg: Write a JavaBeanClass for mobile :

```
public Class Mobile
 {
  private String name;
  private int cost;
  private double version;
  public Mobile(String name, int cost, double version)
   {
    this.name = name;
    this.cost = Cost;
    this.version = version;
   }
  public String getName() {
   return name;
  }
  public void SetName (String Name) {
    this.name = name;
  }
  public int getCost() {
    return cost;
  }
```

25/08/2018
Saturday

```
public void setCost(int Cost){
  this.cost=Cost;
}public double getVersion(){
  return version;
}public void setVersion(double version){
  this.version=version;
}
}
```

MainClass.Java:

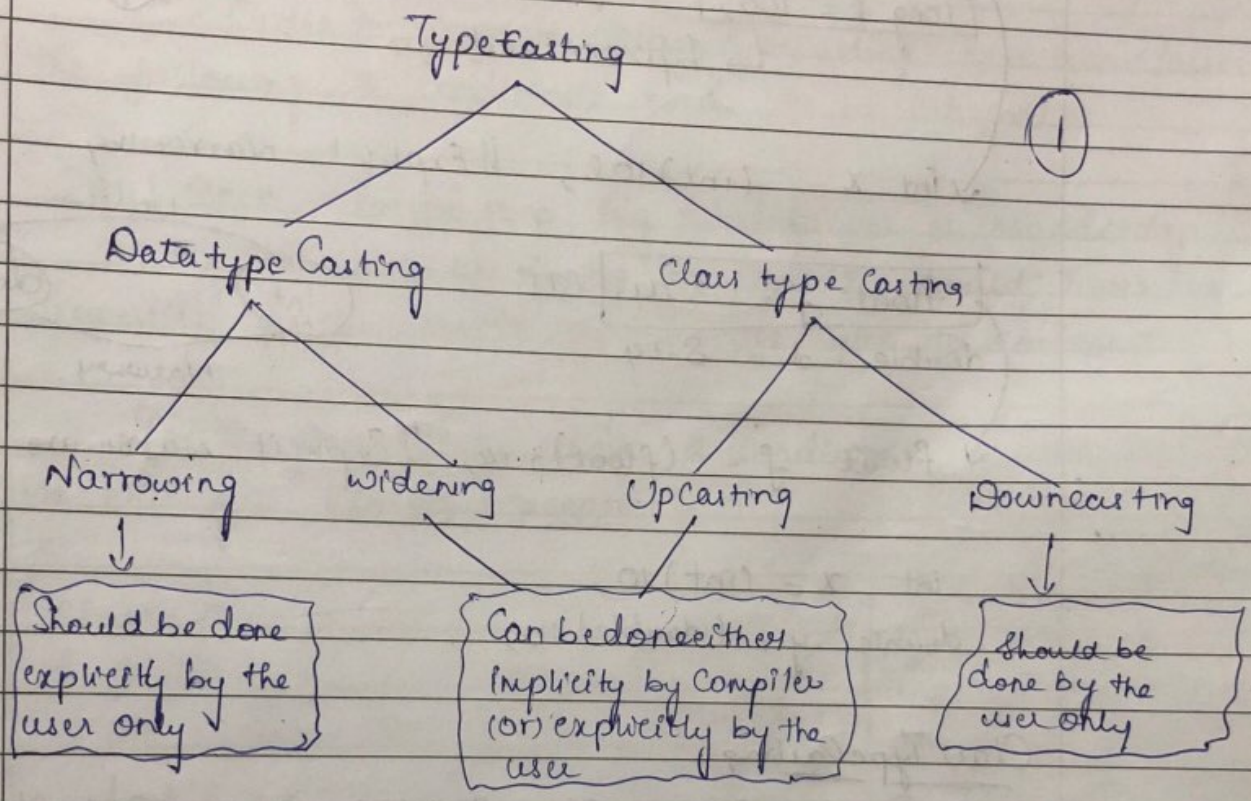
```
Class Mainclass
{psvm(String[] args)
{Mobile M1=new mobile("Apple",17000,11.0);
 Mobile M2=new mobile("Samsung",10000,8.0);
 Sop("first mobile details");
 Sop(M1.name);
 Sop(M1.cost);
 Sop(M1.version);
 Sop("Second mobile details");
 Sop(M2.name);
 Sop(M2.cost);
 Sop(M2.version);
 M1.setVersion(11.18);
 M2.setVersion(8.9);
 Sop("first Mobile details");
 Sop(M1.name);
 Sop(M1.Cost);
 Sop(M1.version);
 Sop("Second mobile details");
 Sop(M2.name);
 Sop(M2.cost);
 Sop(M2.version);
 }
}
```

25/08/18
Saturday

Typecasting:

It is a process of converting one type of information to another type.

Typecasting is of 2 types.



Data typecasting:

1. It is a process of converting one data type of data to another type.
2. Data typecasting is classified into 2 types.
   i) Narrowing → larger to smaller
   ii) Widening → Smaller to larger.

Eg:

```
int x=10;
double y= 10.11;        } type matching
                          Stmts
int x = 10.11;  CTE     } type mismatching
double y = 10;            Stmts
```

→ Implicit widening

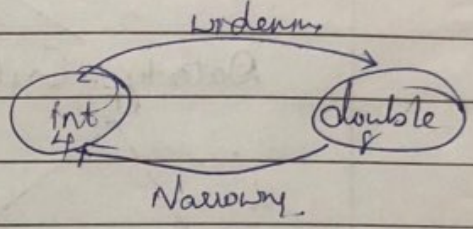
```
int x = (int)10.11; // Explicit Narrowing
int y= (double)10; Explicit Widening
```

S.o.p(x); //10
Sop(y); //100

int x = 10l; → CTE
long l = 100; } Type mismatching starts.
↳ Implicit Widening.

int x = (int) 10l; // Explicit Narrowing.

float f = 3.14 } CTE
double d = 3.14



float f = (float) 3.14; // Explicit Narrowing.

int x = (int) 10;
double y = (double) 10;

Class Typecasting.

1. It is a process of converting from one class type of information to another class. Class Type Casting is classified into following 2 types
   a) Upcasting
   b) Down Casting.

a) Upcasting.

1. Converting from Sub-class type to Super-class type is known as "Upcasting".
2. This can be done either implicitly by the compiler (or) explicitly by the user.
3. When we do upcasting a Sub-class object is refered by a Super class reference. Using the Super class reference we can access only the properties of Super class (Subclass properties will be hidden).

b) DownCasting:

→ Converting from Superclass type to Sub-class type is known as "Downcasting".

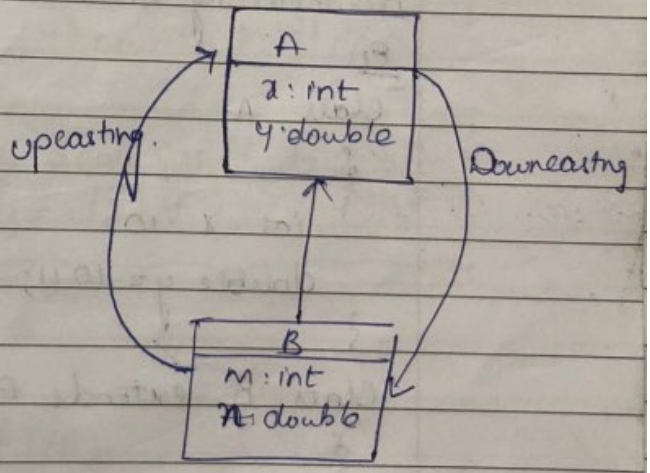
→ This can be done explicitly by the user only.

In Order to perform class typecasting b/w two classes the following 2 conditions needs to be satisfied.

1) Is-A relationship b/w two classes is manditory.

2) The class we are trying to convert should have the properties of the class to which we are trying to convert.

If any one of the above 2 conditions is not satisfied we will get "ClassCastException".

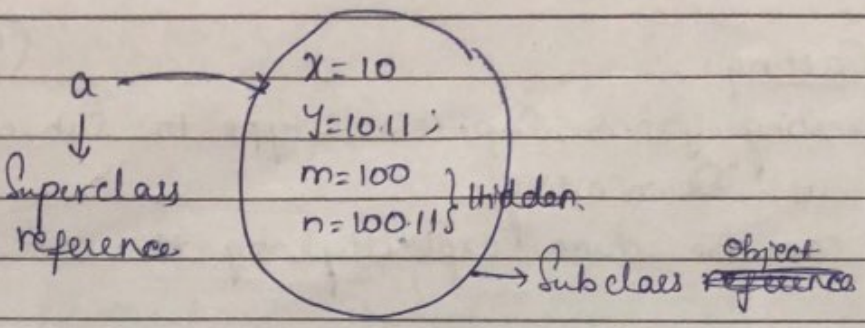


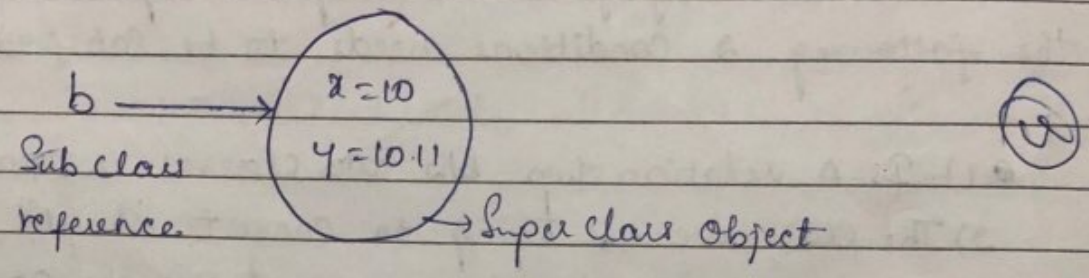

```
Class A
{
  int x=10;
  double y=10.11;
}
Class B extends A
{
  int m=100;
  double n=100.11;
}
A a = new A(); } type matching Stmts
B b = new B();

A a = new B(); → implicit Upcasting
B b = new A(); → CTE
A a = (A) new B(); ⇒ Explicit Upcasting
B b = (B) new A(); ⇒ Explicit downcasting
```

a.x; a.y;

↓

Exp: ClassCastException





Down Casting will not always causes an exception. If we try to perform downcasting without performing upcasting, then only we will get class cast exception. In Otherwords, downcasting should be done only on an upcasted object.

Eg:

```
Class A
{
   int x=10;
   double y= 10.11;
}
Class B extends A
{
   int M=100;
   double n= 100.11;
}
A a= new B();
   a.x;
   a.y;
```

`B b= (B) new A();` → Class cast Exception

`B b = (B) a;` Explicit downcasting.

↓ upcasted

b.m; |
b.n; | ✓

A a = new A();

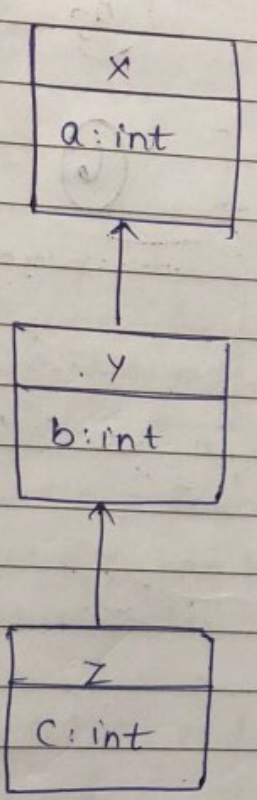
B b = (B) a; explicit downcasting. CCE.

not an upcasted.

a → x=100, y=10+11, m=100, n=100·11

b →

(5)



Case-1:

Z z_1 = new Z(); // Type match
Z z_2 = (Z) new Y(); } CCE
Z z_3 = (Z) new X(); }

Case-2:

Y y_1 = new Y(); // TM.
Y y_2 = new Z(); // Implicit upcasting.
Y y_3 = (Y) new X(); // CCE

Can be down casted to Z type
i.e. Z z_2 = (Z) y_2;

Case-3:

X x_1 = new X(); // TM
X x_2 = new Y(); } Implicit upcasting
X x_3 = new Z(); }

Can be down casted to y type but not Z type
i.e Y y = (Y) x_2; ✓
Z z_3 = (Z) x_2; ✗ CCE

Can be down casted to both Y & Z type. ie
Y y = (Y) x_3;
Z z_3 = (Z) x_3; | ✓

26/08/2018
Sunday.

Case 4:

```
Object O1 = new X();
Object O2 = new Y();
Object O3 = new Z();
```

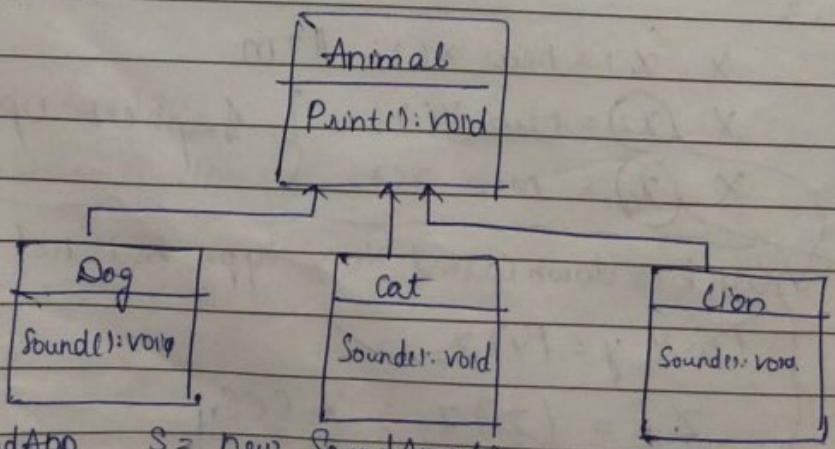
O3 → a, b + object

We can access only Object class properties.

Without typecasting.

```
class SoundApp
{
 public void animalSound (Dog d)
  { d.print();
    d.Sound();
  }
 public void animalSound (Cat c)
  {
    c.print();
    c.Sound();
  }
 public void animalSound (Lion l)
   {
    l.print();
    l.Sound();
   }
}
```

(6)

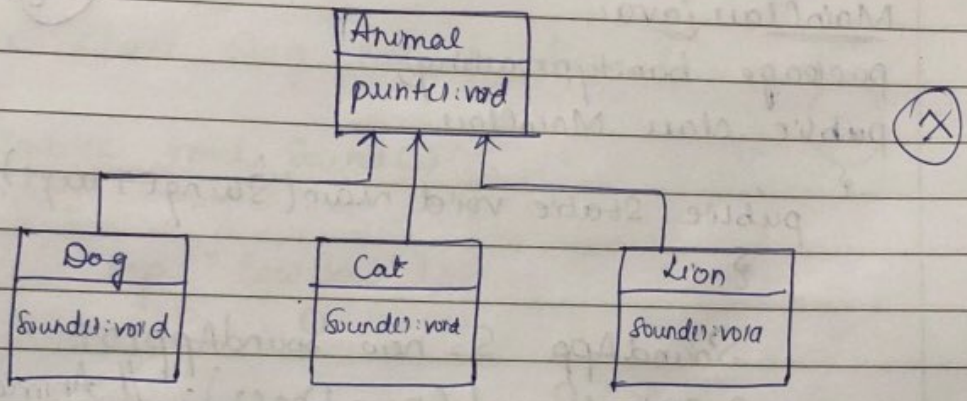



```
SoundApp S = new SoundApp();
S.animalSound (new Dog());  // Dog d = new Dog();
S.animalSound (new Cat());  // Cat c = new Cat();
S.animalSound (new Lion());  // Lion l = new Lion();
```

__/__/____

As shown above, if we don't use class type casting we will end up writing many individual overloaded methods. All these methods are capable of accepting only one type of objects. Hence we call these methods as 'specialized methods'. This approach is known as 'Specialization'.

With type casting,



SoundApp.java

```
package ban.typeCasting;
public class SoundApp
{
  public void animalSound(Animal a)
  {
    a.print();
    if (a instanceOf Dog)
    {
      Dog d= (Dog) a;
      d.Sound();
    }
    else if (a instanceof Cat)
    {
      Cat c= (Cat) a;
      c.Sound();
    }
```

```
        else if(a instanceof Lion)
          {
            Lion l = (Lion)a;
            l.sound();
          }
      }
}
```

MainClass.java:

```
package ban.typecasting;
public class MainClass
  {
    public static void main(String[] args)
      {
        SoundApp S = new SoundApp();
        S.animalSound(new Dog()); // Animal a = new Dog()
        S.animalSound(new Cat()); // Animal a = new Cat()
        S.animalSound(new Lion()); // Animal a = new Lion()
        S.animalSound(new Animal()); // Animal a = new Animal()
// S.animalSound(new Object()); //Animal a = new Object();
      }

  }
```

__/__/____

Animal.java.

```
public class Animal
 {
    public void print( )
      {
       Sop("I am an animal");
      }
 }
```

Dog.java.

```
public class Dog extends Animal
  {
     public void Sound( )
       {
         Sop("Bow Bow");
       }
  }
```

(a)

Cat.java.

```
public class Cat extends Animal
 { public void sound( )
     {
       Sop("meow");
     }
 }
```

Lion.java.

```
public class Lion extends Animal
  {
    public void Sound( )
      {
        Sop("Roar");
      }
  }
```

27/08/2018

Object Obj = new Object(); | Type matching.
String S = new String();

Object Obj = new String(); // implicit Upcasting.
String S = (String) new Object(); CCE

↳ String S = (String) Obj; // Down Casting
↳ Upcasted.

Advantages Of Typecasting:
1. It helps us to achieve generalization.
2. It helps us to achieve dynamic polymorphism through method overriding.
3. It helps us to write generalized catch block.

OVERRIDING:

When we inherit Superclass properties to the Subclass, all the properties of Superclass will be inherited to Subclass. If we are not happy with any of the Superclass method implementation, Subclass can always change the implementation by retaining the same method signature.

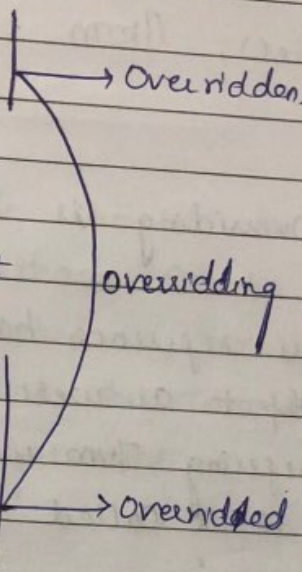
This process of changing the implementation of Superclass method in the Sub-class is known as 'Overriding'.

We Override a method in the Subclass Only if we are not happy with Superclass method implementation.

Eg.

1.

```
class Parent
{
  public void mary()
  {
    Sop("Mary abc");
  }
}
class Child extends Parent
{
  public void mary()
  {
    Sop("mary efg");
  }
}
```



Parent.java

2

```
class Parent
{
  public void getVehicle()
  {
    Sop("Hero Honda");
  }
}
```

Child.java

```
class child extends Parent
{
  public void getVehicle()
  {
    Sop("KTM");
  }
}
```

MainClass.java

```
public class Mainclass
{
  P.S.V.M(String[] args)
  {
```

```
Parent P1 = new Parent();
P1.getVehicle(); //Hero Honda
Parent P2 = new Child();
P2.getVehicle(); //KTM
}
}
```

Justify how Overriding is Dynamic Polymorphism.

Superclass reference has the capability of refering to superclass Object as well as Sub.class Object. To which Object it is refering JVM will come to know during runtime. Hence it is called "dynamic Polymorphism." (or) "Runtime Polymorphism".

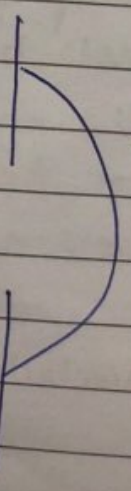
IMPORTANT CONCLUSIONS.

Case-1:

While overriding a method in a subclass either we should retain the same access modifier or increase the visibility (but we can't decrease the visibility). Private methods cannot be overriden beeoz they donot inherit.

Eg:

```
class A
{
  private void m1()
  {
    E
  }
}
class B extends A
{
  private void m1()
  {
    E
  }
}
```



looks like overriding but it is not overriding.

Parent	public	protected	default	private
Child	public ✓	protected / public ✓	default / public / protected ✓	

Case-2:

Abstract methods should be overridden in the Sub-class. Concrete methods can be overridden as concrete and also as abstract. When we override concrete method as abstract, We can prevent the concrete method implementation for all the further Sub-classes.

Eg.

```
abstract class Parent
  {
    abstract public void m1();
    public void m2()
      {
        Sop ("Hello world");
      }
  }

abstract class child extends Parent
  {
    public void m1()
      {
        ≡
      }
    abstract public void m2();
  }
```

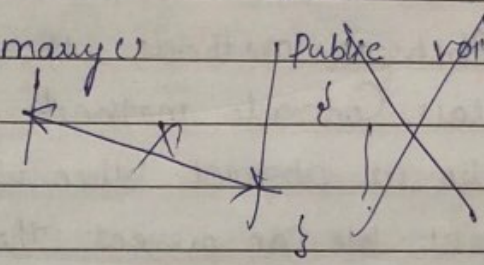
Parent :	abstract	Concrete
Child :	abstract / Concrete ✓	Concrete / abstract ✓

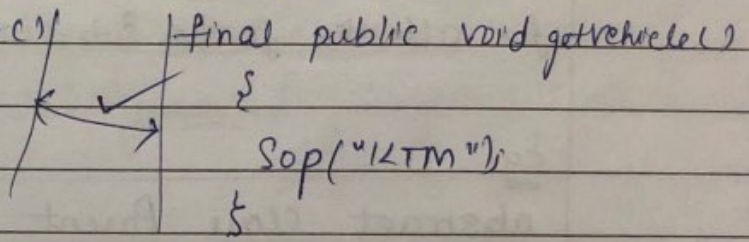
Case-3:

final methods will be inherited to the subclass but cannot be overridden in the sub-class.

Eg:

```
class Parent                          class Child extends Parent
{                                     {
  final public void m1()                public void m1()
  {                                     {
    Sop("abc");                         }
  }
  public void getVehicle()            final public void getVehicle()
  {                                   {
    Sop("Hero Honda");                  Sop("KTM");
  }                                   }
}
```





Parent : final — non-final

Child : ✗ — non-final / final ✓

final var : Can't reassign
final class : Can't inherit from
final method : Can't override.

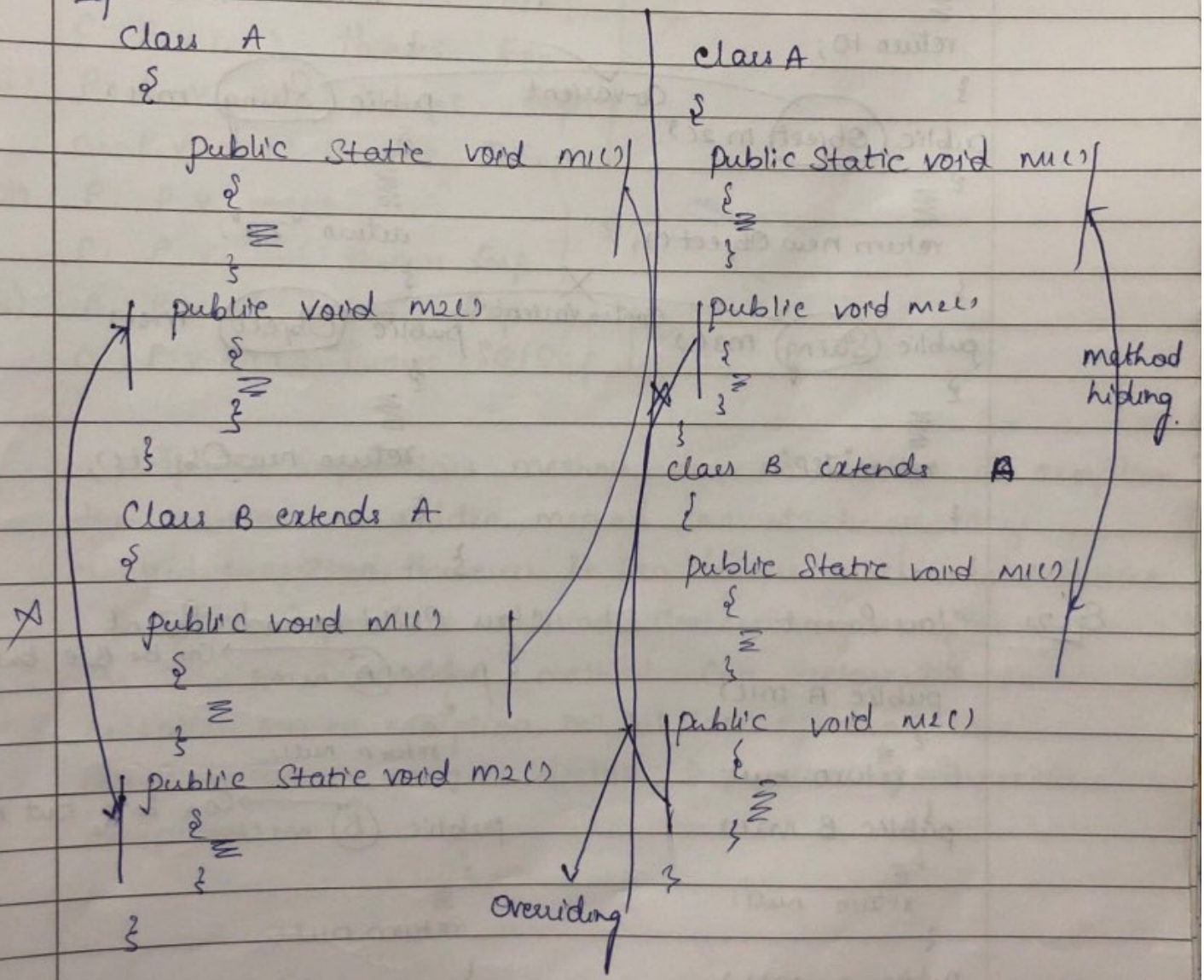
28/08/2018
Tuesday.

Case-4:

'Static' methods are class-level methods which cannot be overridden in the subclass. When we write static method in both super class and subclass with same signature, then it looks like overriding but it is not overriding. It is called by a new name "METHOD HIDING".

'Static' methods cannot be overridden as non-static and vice-versa.

Eg:



Note:

Since main method is static, which cannot be overridden

__/__/____

Imp Case-5: (Return type)

While Overriding a method in the Subclass we cannot change the return type, if the return type is primitive. If the return type is class type then 'covarient' types can be changed but not 'contravarient'.

Eg-1

```
class A
{
public int m1()
{
return 10;
}
public Object m2()
{
return new Object();
}
public String m3()
{
return "JSP";
}
}
```

```
class B extends A
{
public int m1()
{
return 100;
}
public String m2()
{
return "JSP";
}
public Object m3()
{
return new Object();
}
}
```

not allowed to change

Covarient

Contravarient

Eg-2

```
class Parent
{
public A m1()
{
return null;
}
public B m2()
{
return null;
}
public C m3()
{
return null;
}
}
```

```
class Child extends Parent
{
public A m1()
{
return null;
}
public B m2()
{
return null;
}
public C m3()
{
return null;
}
}
```

Can be B, C but not Object

Can be C, but not A & Object

Can't be B/A/Object

15/09/2018
Saturday

Case-6 (Applicable Only for Checked):

1) Parent : P.V.m1()
   Child : P.V.m1() throws IOExp | ✗

2) P: P.V.m1() throws IOExp
   C: P.V.m1() | ✓

3) P: P.V.m1() throws EOFExp
   C: P.V.m1() throws IOExp | ✗

4) P: P.V.m1() throws IOExp
   C: P.V.m1() throws FileNotFoundExp | ✓

5) P: P.V.m1() throws IOExp
   C: P.V.m1() throws Exp | ✗

6) P: P.V.m1() throws Exp
   C: P.V.m1() throws IOExp | ✓

7) P: P.V.m1()
   C: P.V.m1() throws Exp | ✗

8) P: P.V.m1() throw Exp
   C: P.V.m1() throws SQLExp | ✓

1. If the Superclass method does not declare an exception, the Subclass overridden method can not declare any checked exception. However it can declare unchecked exception.

2. If the Superclass method declares any exception then the Subclass Overridden method can declare the same exception (or) no-exception (or) Subclass of Superclass exception. But it cannot declare Superclass of Superclass exception.

__/__/____

Diff b/w Overloading & Overriding:

Overloading	Overriding
1. If we want to perform one task in multiple ways then we will go for Overloading.	1. If we want to change the task itself then we will go for Overriding.
2. It is a process of writing multiple methods with the same name but different signature.	2. It is a process of changing the implementation of Super-class method in the Sub-class.
3. Is-A relationship is not mandatory.	3. Is-A is relationship is mandatory.
4. Static / final / Private methods can be Overloaded.	4. Static / final / Private methods cannot be Overridden.
5. It is an example for Compile time polymorphism.	5. It is an example for dynamic polymorphism.
6. Constructors can be Over-loaded.	6. Constructors cannot be Overridden.

Polymorphism:

1. An Object showing different states in its different stages of life is known as 'polymorphism'.

2. Polymorphism can be classified into following 2 types:

a) Compile time polymorphism
b) Run-time polymorphism

a) Compile time polymorphism.

It is also known as Static polymorphism. In this type of polymorphism binding of method declaration with method definition is done by the Compiler during compilation time. Hence they name Compiletime polymorphism. It does the binding based one one of the following parameter.

i) Length of argument
ii) Type of argument
iii) Order of occurance of argument.

Eg: Method Overloading, Constructor Overloading, method hiding.

b) Run-time polymorphism:

It is also known as dynamic polymorphism. In this type of polymorphism binding of method declaration with the method definition is done by the Jvm during Runtime based on the 'Runtime Object' and hence the name runtime polymorphism.

Eg: Method Overriding

Section-III

Object Class:

1. It is the Supermost class in the entire java hierarchy.
2. It is available in java.lang Package.
3. It has not argument Constructor only.

```
Object O1 = new Object();   ✓
Object O2 = new Object(10); ✗
```

Methods of Object Class:

||types || methods

1. public String toString()
2. public int hashCode()
3. public boolean equals(Object obj)
4. public final void wait()
5. public final void wait(long ms)
6. public final void wait(long ms, int ns)
7. public final void notify()
8. public final void notifyAll()
9. public Class getClass()
10. protected void finalize()
11. protected Object clone()
12. private void regristerNative()

Related to Threads, which can't be overridden

4, 5, 6 } overloaded methods.

1. public String toString():

It returns string representation of an object. String representation includes the fully qualified name of the class along with the hashcode in hexa-decimal in the following format.

PkgName.ClassName@hashcode in hexadecimal.

29/08/18
Wednesday

Usually we override this method in the Sub-class to display states of an Object (Contents of an Object).

2. public int hashCode():

Hashcode is a unique integrated integer number associated with an Object. This method generates the hashcode based on hexa-decimal address.

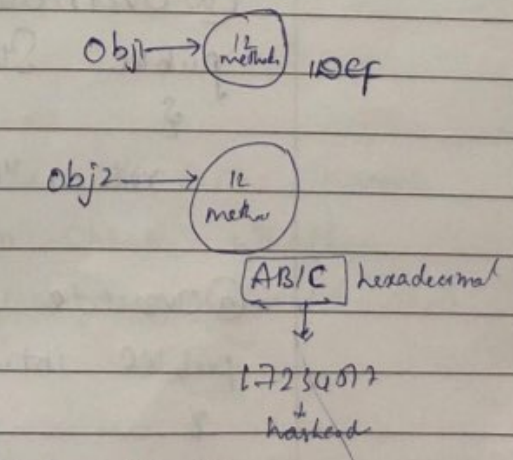
Usually we Override this method in the Sub-class to generate the hashcode based on unique attribute of an Object.

3. public boolean equals (Object Obj):

This method is used to compare the Current Object with the given Object based on the hash-code. If two Objects are having same hash code then it returns true Otherwise false.

Usually we Override this method in the Sub-class to compare the Current Object with the given object based on the States of an Object.

ObjectDemo1.java




```
package ban.objectclass;
public class ObjectDemo1
{
    P.S.V.M(String[] args)
    {
        Object obj1 = new Object();
        Object obj2 = new Object();
        Sop(obj1); // toString() automatically.
        Sop(obj2);
        Sop(obj1.hashcode());
        Sop(obj2.hashcode());
        Sop(obj1.equals(obj2));
    }
}
```

__/__/____

NOTE:

When we use a reference inside System.out.println() internally it calls toString() Only.

Eg:

Mobile.java.

```
package ban.Objectclass;
public class Mobile
{
    String name;
    double prize;
    int ram;

    public Mobile(String name, double price, int ram)
    {
        this.name = name;
        this.price = price;
        this.ram = ram;
    }
@Override → used to check whether we are overriding correctly or not
    public String toString()
    {
        return "Name=" + this.name + " Price = " + this.price +
                Ram = " + this.ram + "GB";
    @Override
    public int hashcode()
    {
        return this.ram;
    }
    @Override
    public boolean equals(Object obj)
    {
```

__/__/____

```
Mobile m= (Mobile)obj; // down Casting.
return this.name.equals(m.name) &&
       this.price == m.price &&
       this.ram == m.ram;
}
```

ObjectDemo2.java.

```
package ban.objectclass;
public class ObjectDemo2
{
  public static void main(String[] args)
  {
  Mobile m1= new Mobile("Samsung", 15000.00, 2);
  Mobile m2 = new Mobile("Samsung", 15000.00, 2);
  Sop(m1);
  Sop(m2);
  Sop(m1.hashCode());
  Sop(m2.hashCode());
  Sop(m1.equals(m2));
  }
}
```

Assignment:

1. Create a class called car with the attributes name, color and price. Write a program to check whether both the cars are same not. The program should also display the details of the car.

2. Write a program to check whether the time in 2 watches is same or not. The program should also display the current object time. (Override two strings)
   HH:MM:SS.

30/08/2018
Thursday.

Diff b/w equals Operator & equal method.

1. 'equals' Operator is used to compare two references based on actual address
2. 'equals method' default implementation is still to compare Objects based on actual address.
3. If we do not override equals method then there is no difference b/w equals operator and equals method.
4. It is recommanded practice to override equals method and hashcode method together.

⇒ Create an array of Pizza Objects and display the Pizza Objects whose cost is more than 300.

```
P.S.V.M(--
  {
   Scanner sc = new Scanner(System.in);
   Sop("Enter the Array Size");
   int Size = Sc.nextInt();
   Pizza[] arr = new Pizza[Size];
   Sc.nextLine();
   for(int i=0; i<arr.length; i++)
   {
      Sop("Enter the name");
      String name = sc.nextline();
      Sop("Enter the size");
      String Psize = Sc.nextline();
      Sop("Enter the cost");
      double cost = Sc.nextDouble();
      Sc.nextLine();
      Pizza p = new Pizza(name, Psize, cost);
      arr[i] = p;
}
```

31/08/2018
Friday.

```
Sop("Display All pizza Object");
for(int i=0; i<arr.length; i++)
  {
    Sop(arr[i]);
  }
Sop("Pizza Greater than 300");
for(int i=0; i<arr.length; i++)
{
   Sop(arr[i]);
}.
```

Assignment

→ Create a class called ~~Car~~ Student with the attributes name, id and marks. Create an array of 5 student Objects and read the ~~Stude~~ information from the user.
Write a program to display all the Students whose marks is greater than 60. write a program to display highest marks student. Write a prgm to display all the Student whose name Starts with 'S'.

→ Create a class called book with the attributes name, author and price. Create an array of 5 book Objects and do the following Operation.
→ Display all the book details
→ Display all the books whose price is greater than 250.
→ " " " " written by Padmareddy.
→ " " " java books written by different authors.

02/09/2018
Monday.

getClass():

It returns an object of class class. Using that object we can invoke getName() method which returns the fully qualified name of the object.

Class → keyword.

Class.java → built-in class.

```
class Class
{
    getName()
    {
    }
    }
}
```

```
class Object
{
    public class getClass()
    {
        return new Class();
    }
}
```

```
Class c = C.getClass();
String s = C.getName();
Sop(s);
```

`C.getClass().getName();`

String Class:

1. It is a final class available in java.lang package.
2. String is immutable class.
3. String implements Comparable interface. Hence array of String objects can be sorted.
4. String is the only class in the entire java whose object can be created both by using 'new' and without using 'new'.
   i) String S = new String("Hello"); → With 'new'
   ii) String S = "Hello"; ← → without 'new'.

5. The following 3 methods of Object class overridden in the String class.

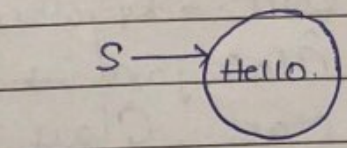
i) toString():

This method has been overridden to display the states (Contents) of an Objects.

String S = "Hello";

Sop(s); // Hello

↳ toString() of string class called



ii) public int hashCode():

This method has been overridden to generate the hashcode based on the states (Conents). If two objects are having same state then the hashcode will be same otherwise different.

iii) public boolean equals (Object obj):

This method compares the current object with the given object based on the states (Contents) of an objects. If two objects are having same contents it returns true otherwise false.

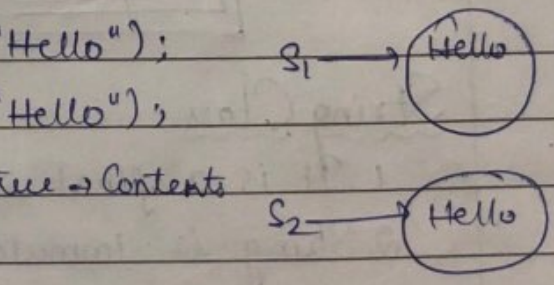
String S_1 = new String ("Hello");

String S_2 = new String ("Hello");

Sop (S_1.equals(S_2)); // true → Contents

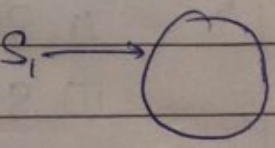
Sop (S_1 == S_2); // false

Reference Comparision

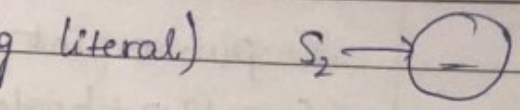


Constructors of String Class.

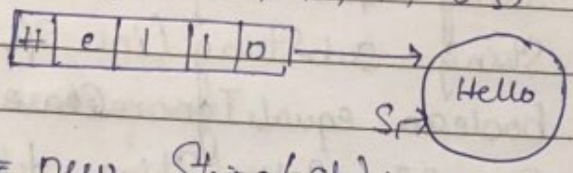
1. String S_1 = new String ();



2. String S_2 = new String (String literal)

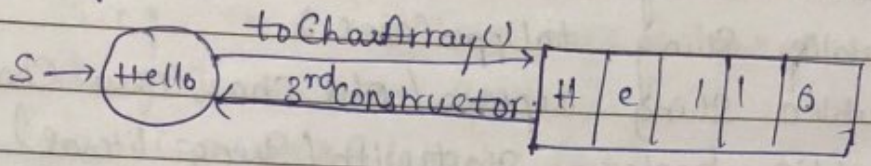


Char[] ch = {'H', 'e', 'l', 'l', 'o'};



3. String S = new String(ch);

String S = "Hello";



Program:

```
package ban.StringClass;
public class StringDemo1
{
  P.S.v.m(String[] args)
  {
    String s1 = new String("Hello");
    String s2 = new String("Hello");
    Sop(s1); //Hello
    Sop(s2); // Hello.
    sop(s1.hashcode()); // Same hashcode
    Sop(s2.hashcode()); // Same hashcode
    Sop(s1.equals(s2)); //true
    Sop(s1 == s2) // false
```

String Class Methods:

1. public int length()
2. public char charAt(int index)
3. public char[] toCharArray()
4. public int indexOf(Char ch)

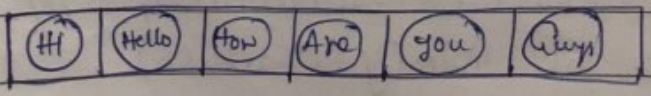
5. public int lastIndexOf(Char ch)
6. public boolean contains(String literal)
7. public String subString(int sp)
8. public String subString(int sp, int ep) // ep-1
9. public boolean equalsIgnoreCase(Object obj)
10. public String[] Split(String delimeter)
11. public String concat(String str)
12. public String toLowerCase()
13. public String toUpperCase()
14. public String replace(old char, new Char)
15. public boolean Startswith(String literal)
16. public boolean endswith(String literal).

04-09-2018
Tuesday.

StringDemo.java.

```
package ban.StringClass;
public class StringDemo
{
  public static void main(String[] args)
  {
    String str = "Hi Hello How are you guys";
    String[] s = str.split(" ");
    Sop("Total Words= "+s.length);
    for(int i=0; i<s.length; i++)
    {
     Sop(s[i]+ "contains " +s[i].length()+ "Char's");
    }
  }
}
```

After Splitting.



s.length ; //6
s[0].length(); //2

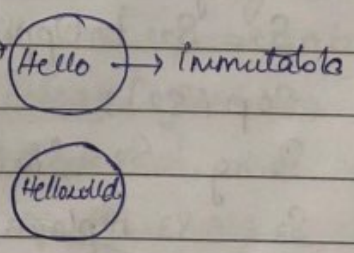
→ Count no of occurance of 'love' in given Sentence.
String s= "I Love java I love c++ I love c";

Program1.java:

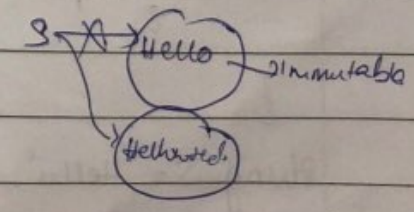
```
public class Program1
{
  P.S.V.M(String[] args)
  { int count=0;
    String str = "I Love java I love c++ I Love c";
    String[] s = str.split(" ");
    for(i=0; i<s.length; i++)
    {
      if(s[i].equalsIgnoreCase("love"))
      {
        int count.count++;
      }
    }
  S.O.p("Total no. of occurance"+count);
}
```

Immutability:

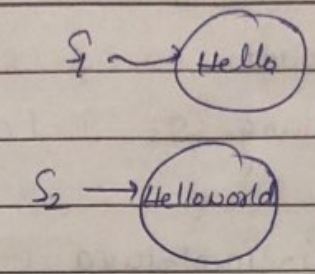
```
String s="Hello";
s.concat("world");
Sop(s); // Helloworld x
        // Hello. ✓
```




```
String s="Hello";
  s= s.concat("world");
  sop(s); //Helloworld
```



```
String S1 = "Hello";
String S2 = S1.Concat("world");
Sop(S1); // Hello
Sop(S2); // HelloWorld
```



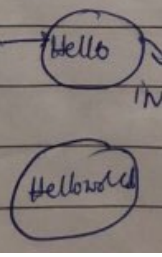
Immutability. (Justify how string is immutable):

Whenever we create an object of String class we cannot peform any changes in that object. If we try to perform any changes, a new object will be created with those new changes. This unchangable behaviour is known as Immutability.

StringDemo4.java

```
public class StringDemo4
{
    PSVM(String[] args)
    {
        String S1 = "Hello";
        S1 = S1.Concat("world");
        Sop(S1); // Helloworld.
        String S2 = "hello";
        S2 = S2.toUppercase(); //HELLO
        Sop(S2);
        String S3 = "hello";
        S3 = S3.replace('h', 'H');
        Sop(S3); // Hello
    }
}
```

```
String S = "Hello";
S.Concat("World");
Sop(S); // Hello
```

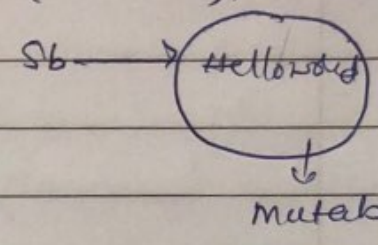


StringBuffer sb = "Hello"; ✗

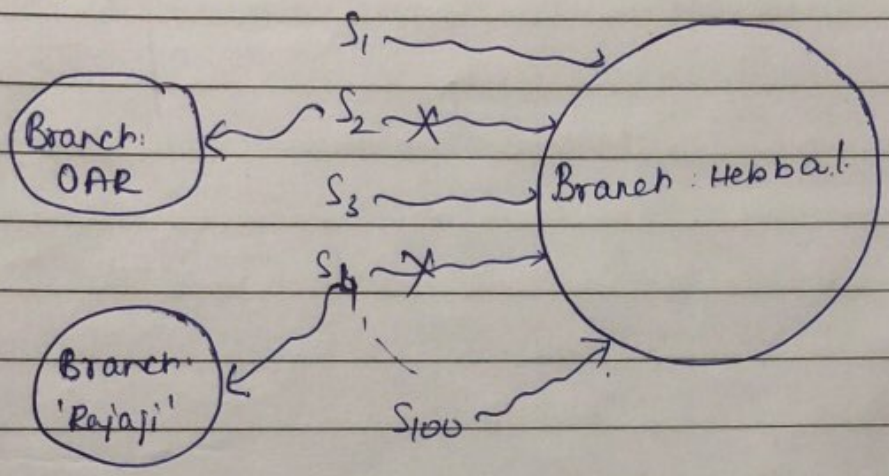
StringBuffer sb = new StringBuffer("Hello");

sb.append("World");

Sop(sb); //HelloWorld



* Why String is immutable?



S_3.branch = "OAR";

S_4.branch = "Raj";

When a String object is refered by multiple references using one reference if we try to change the content, that should not affect the other references. This is possible only if the object is immutable. Hence String is made as immutable.

Since String is immutable, every time we try to change the content we will end up creating a new object. This might leads to "memory leak" problem. To address this memory leak issue java introduced two more classes related to string and both of them are mutable.

i) StringBuffer } mutable

ii) StringBuilder }

05/09/2018
Wednesday

String	StringBuilder	StringBuffer
1. It is a final class available in java.lang.	1. java.lang & final class	1. java.lang & final class.
2 String is immutable	2. Mutable.	2. Mutable
3 It implements comparable interface. Hence array of String objects can be sorted.	3. It doesn't implement Comparable interface. Hence array of String builder objects cannot be sorted.	3. It doesn't implemen Comparable interfac Hence array of String buffer objects cann be sorted.
4 Methods are Not-Synchronized.	4 Methods are not Synchronized.	4. Methods are Synchro-nized.
5. It is a thread Safe.	5. It is not a thread safe.	5. It is a thread safe.
6. Three methods of Object class overridden in the String class. i) toString ii) equals iii) hashcode.	6. Only tostring() method has been Overridden.	6. Only toString() method has been Overridden.
7. Object can be created both by using new and without using 'new'.	7. Objects should be created using 'new' only.	7 Objects Should be Created using 'new' only.

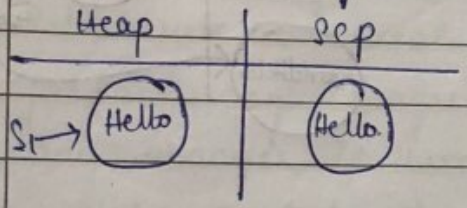
→
```
String S1 = new String("Hello");
String S2 = new String("Hello");
S1.equals(S2); // true
S1==S2;        // false
Sop(S1);
Sop(S2);       } Hello
```

→ StringBuffer Sb1 = new StringBuffer ("Hello");
StringBuffer Sb2 = new StringBuffer ("Hello");
Sb1.equals(Sb2); // false
Sb1 == Sb2; // false.
Sop(Sb1); } Hello.
Sop(Sb2); }

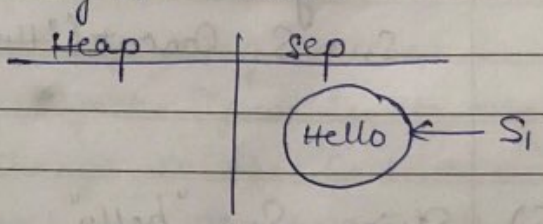
(Sep)

String Constant Pool Vs Non-Constant Pool (Heap):

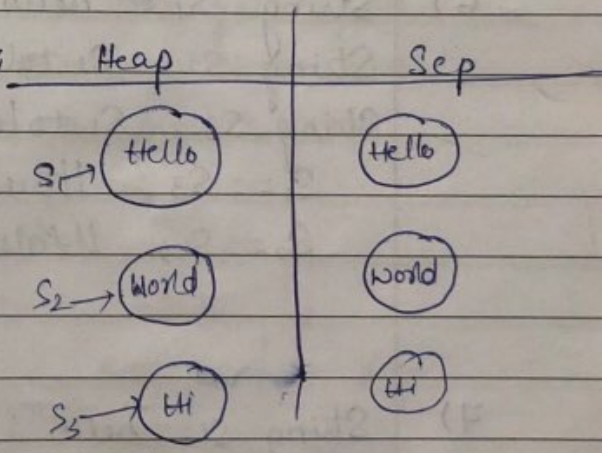
1) String S1 = new String ("Hello");



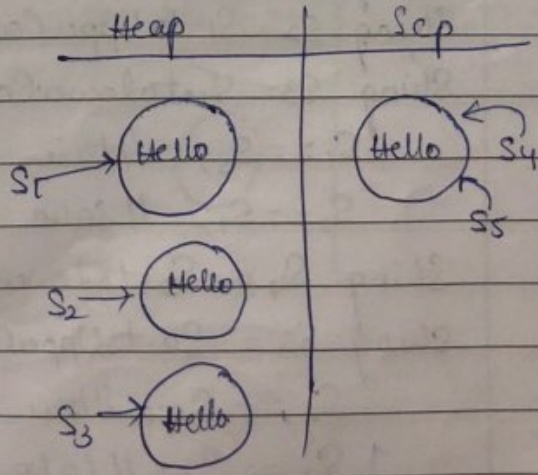
String S1 = "Hello";



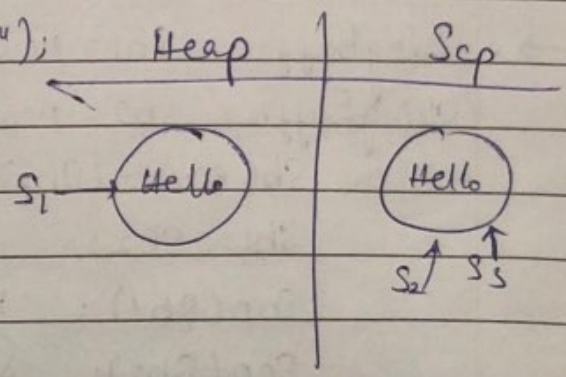
2) String S1 = new String ("Hello");
String S2 = new String ("World");
String S3 = new String ("Hi");



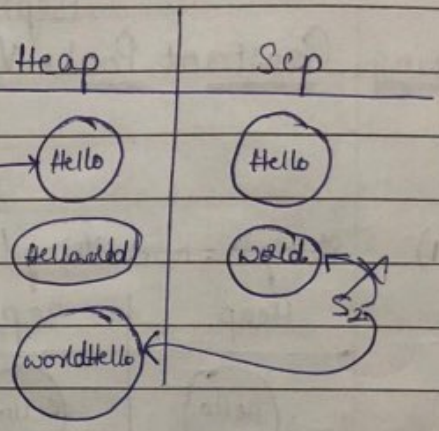
3) String S1 = new String ("Hello");
String S2 = new String ("Hello");
String S3 = new String ("Hello");
String S4 = "Hello";
String S5 = "Hello";
S4 == S5 // true.



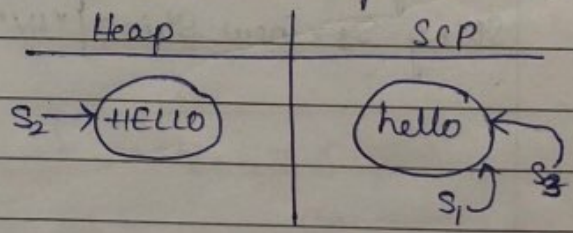
4) String S_1 = new String("Hello");
String S_2 = "Hello";
$S_1 == S_2$; // false
S_1.equals(S_2); // true
String S_3 = "Hello";
$S_2 == S_3$;
S_2.equals(S_3) } true



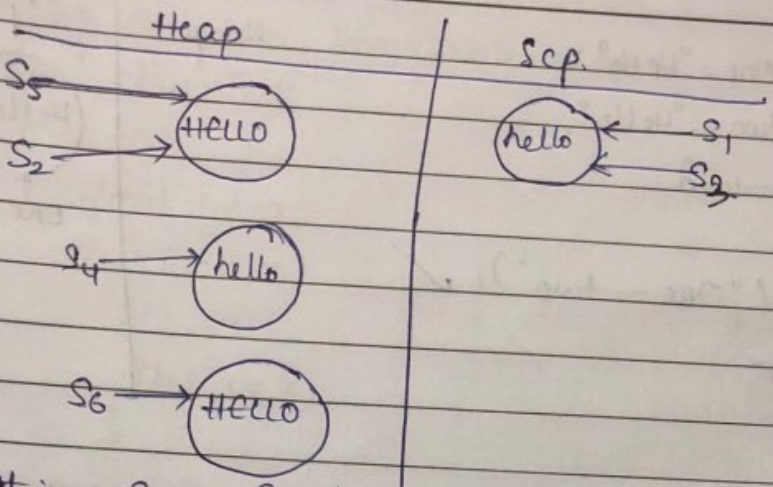
5) String S_1 = new String("Hello");
S_1.concat("world");
String S_2 = "world";
S_2 = S_2.concat("Hello");



6) String S_1 = "hello";
String S_2 = S_1.toUpperCase(); // there is change in content
String S_3 = S_1.toLowerCase(); // there is no change in the content
$S_1 == S_3$; // true
$S_2 == S_1$; // false



7) String S_1 = "hello";
String S_2 = S_1.toUpperCase(); // there is change in the content.
String S_3 = S_1.toLowerCase(); // there is no change in the content
$S_1 == S_3$; // true
$S_2 == S_1$; // false
String S_4 = S_2.toLowerCase(); // Change in the content
String S_5 = S_2.toUpperCase()
$S_2 == S_5$; // true
$S_2 == S_4$; // false.

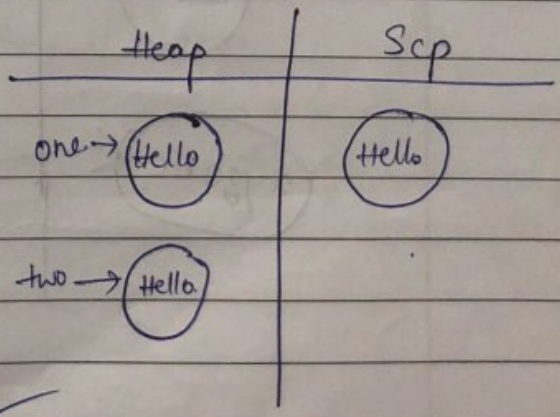


String S6 = S4.toUpperCase();

Immutability (Re Definition):

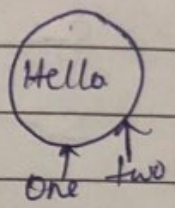
Whenever we create an Object of String class we cannot perform any changes in that Object. If we try to perform any changes, if there is a change in the Content a new Object will be created with those new changes. If there is no change in the Content then existing Object will be reused. This behaviour is known as immutability.

Ex-1:
```
String one = new String("Hello");
String two = new String("Hello");
if(one==two)
{
  Sop("one==two");
}
else
{
  Sop("one!=two");
}
```



2.
```
String one = "Hello";
String two = "Hello";
if (one == two)
  {
    Sop ("One == two");
  }
else
  {
    Sop("One != two");
  }
```

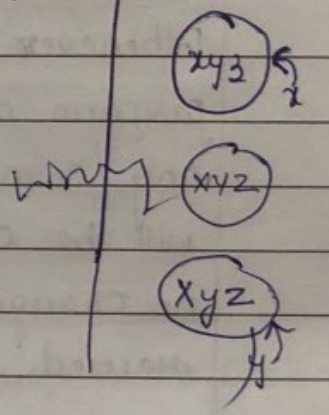
Heap | Scp



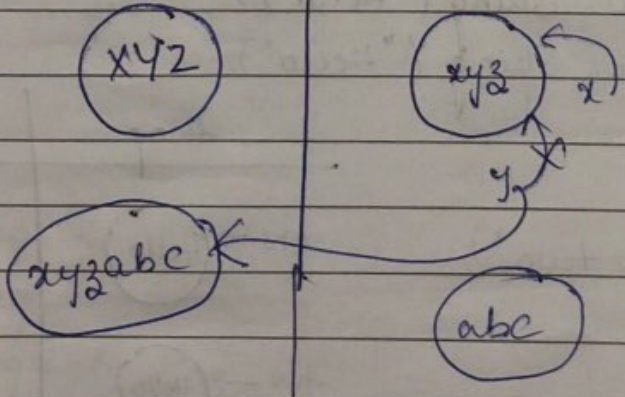
3.
```
String x = "xyz";
x.toUpperCase();
String y = x.replace('Y', 'y');
y = y + "abc";
Sop(y);
```

XYZabc

Heap | Scp.



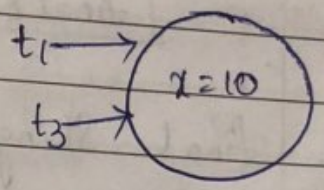
Heap | Scp

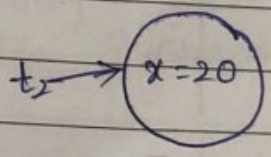


06/09/2018
Thursday.

Writing Our Own Immutable class:

```
final class Test
 {
   final int x;
   Test(int x)
    {
     this.x = x;
    }
   public Test modify(int x)
    {
     if(x != this.x)
      {
       return new Test(x);
      }
     else
      {
       return this;
      }
    }
 }
```






```
Test t1 = new Test(10);
Test t2 = t1.modify(20); // there is change.
                         Runtime operation
Test t3 = t1.modify(10); // there is no change
  Sop(t1==t2); // false
  Sop.(t1==t3); // true
```

Final Versus Immutable:

1. final String S_1 = "Hello";

   S_1 = S_1.concat("world");

   ✗

   final & immutable

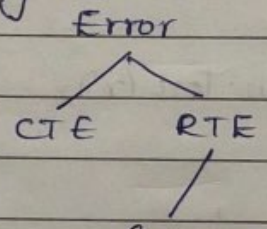
2. final StringBuffer Sb_1 = new StringBuffer("Hello");

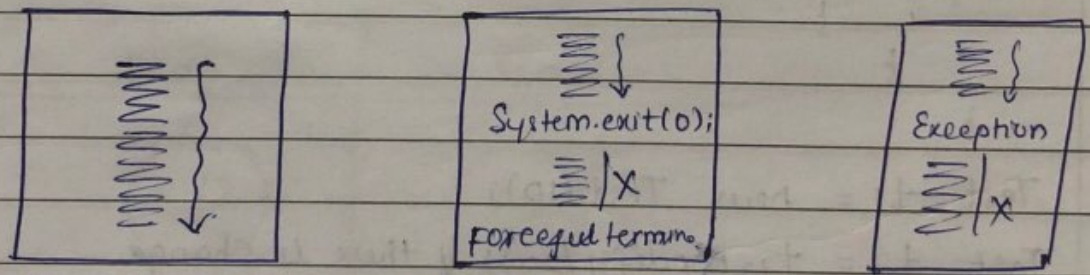
   Sb_2 = Sb_1.append("world");

   ✓

   final & mutable

✱ Exception Handling:





Normal termination | forceful termination | Abnormal Termination.

Exception is an unwanted, unexpected event which disturbs the normal flow of execution.

(or)

It is an unexpected & unwanted event triggered in the JVM, which makes the JVM to terminate the program abnormally.

07/09/2019
Friday.

```
Class A                    Class B {                  Class {
{ P.S.V.m(__)              P.S.V.m(__)                P.S.V.m(_)
  { Sop("mms");            { Sop("MMS");              {
    int x=10, y=5;           int x=10, y=5;             Sop("mms");
    Sop(x/y);                Sop(x/y);                  int x=10, y=0;
    Sop("MME");              System.exit(0);            Sop(x/y);
  }                          Sop("mme");                Sop("mme");
}                          }                          }
Normal Termination         Forceful Termination       abnormal Terminal
                                                      (abnormal line)
```

Default Exception Handler:

When exception occures in a program, JVM will create an object of respective type of exception and throws it back to the program. If user has not written any code to handle the thrown object then JVM calls an assistant called "Default Exception Handler" to handle that object.

Default exception handler will first terminate the program abnormally and display the information about the exception to the user.

ExceptionDemo1.java:

```
public class ExceptionDemo1
{
  P.S.V.m(S__)
  {
    Sop("Main method Started");
    int x=10, y=0;
    Sop(x/y); //exception line
    Sop("Main method ended"); // this will not be executed.
  }
}
```

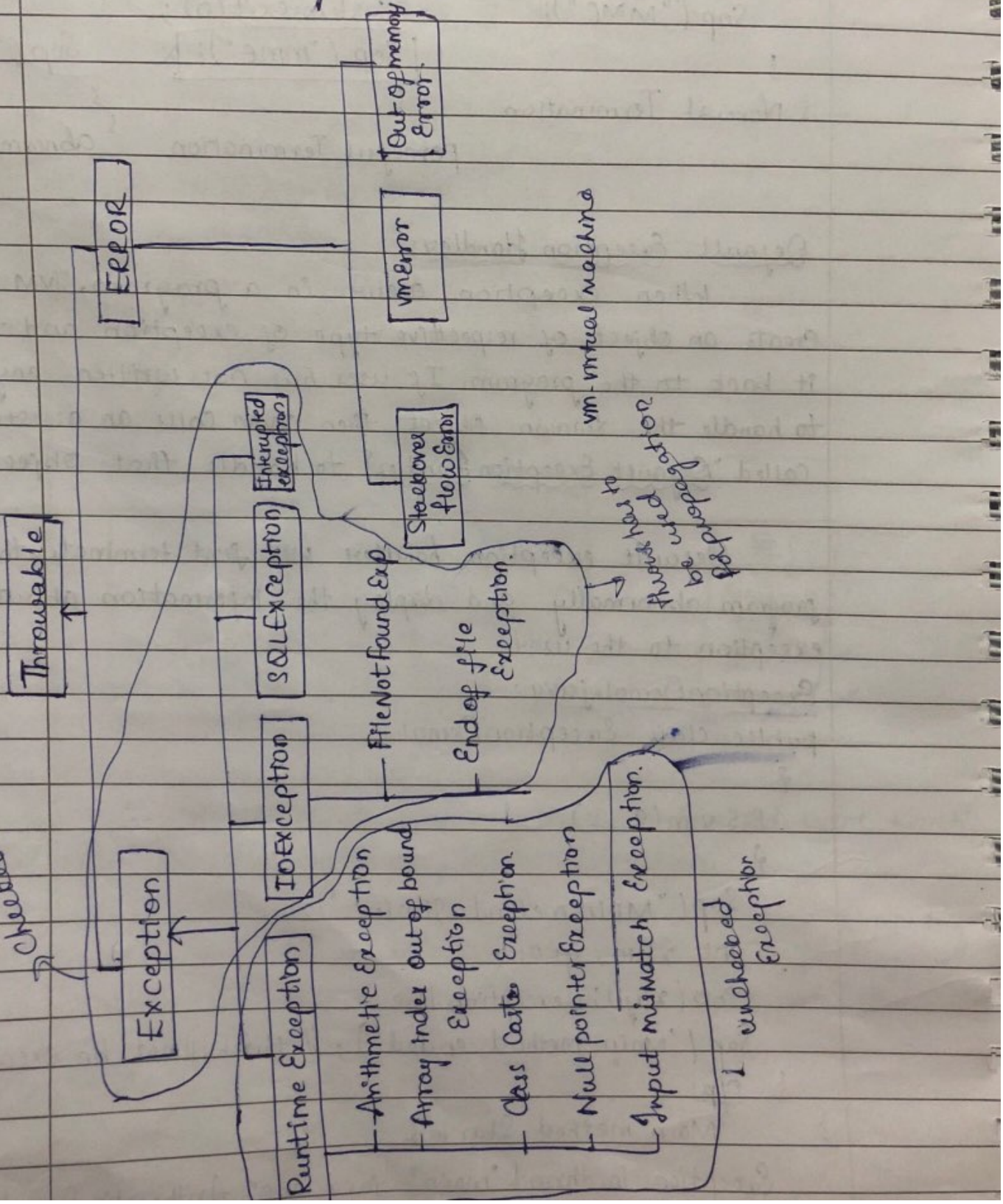
O/P:

Main method Started

Exception in thread "main" java.lang.ArithmeticException: / by Zero at ban.exception.ExceptionDemo1.main(ExceptionDemo1.java:4)

If default exception handler handles the exception then it will terminate the program abnormally. If we want the normal termination then user should handle the exception.

Exception hierarchy in Java:



__/__/____

Handlers:

With the help of following handlers user can handle the exception.

* try
* Catch
* throw
* throws
* finally.

1) try - Catch

```
try
{
    // Suspected line of code
}
Catch(_ _ _)
{
    {
    }
}
```

'try' block Contains only those lines of code which might cause an exception. When the exception occurs in the try block JVM will create an object and throws it back to the programer or user. User should have written appropriate Catch block to Catch the object thrown from the try block. If the exception occurs in the try block only then Catch block will be executed.

Once the Control leaves from the try block, it will never come back to try block again. Due to this all the lines of code which is written below the exception will not be executed.

08/09/2018
Saturday.

Just bcoz we have written try block it doesn't mean exception will always occur.

Example:

```
P.S.V.m(String[] args)
{
  Sop("Main method Started");
  int x=10, y=0;
  try
  {
    Sop("entering try block");
    Sop(x/y); //explina
    Sop("exiting try block"); // this will not be executed
  }
  Catch(ArithmeticException e)
  {
    Sop("Entering Catch block");
    Sop(e.getMessage());
    Sop("exiting Catch block");
  }
  Sop("Main method ended");
 }
}
```

In one try block utmost one exception can occur. When the exception object is thrown from the try block that object will be caught in the respective catch block.

If the user has not written the respective catch block then JVM will handle the exception.

A try block can have multiple catch blocks. Depending upon the type of exception occured, the

try Only one catch block will be executed but not all.

try with multiple Catch blocks:

Example-1

```
package ban.exceptions;
public class ExceptionDemo1
{
    P.S.V.m (String[] args)
    {
     Sop ("MMS");
     int x=10, y=0;
     int[] a= new int[5];
     try
     {
     Sop ("Entering try block");
     Sop (x/y); //exp occured
     a[5]=100; //though it's exp it will not occur
     Sop ("Exiting try block");
    }
    Catch(ArrayIndexOutOfBoundException e)
    {
     Sop("Entering catch block AIOBE");
     Sop (e.getMessage());
     Sop("Exiting Catch block AIOBE");
    }
    Catch (ArithmeticException e)
    {
     Sop ("Entering AE");
     Sop(e.getMessage());
     Sop ("Exiting AE");
    }
```

```
Catch (NullPointerException e)
  {
    Sop("Entering Catch block NPE");
    Sop(e.getMessage());
    Sop("Exiting Catch block NPE");
  }
  Sop("Main method ended");
  }
}
```

Example-2

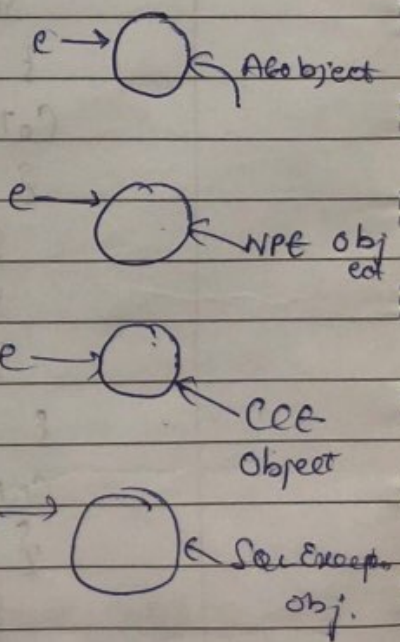
```
try
{
  ≡
}
Catch(AE e)
  {≡
  }
Catch(NPE e)
  {≡
  }
Catch(AIOBE e)
  {≡
  }
Catch(FileNotFound Excep e)
  {≡
  }
Catch(SQLExcep e)
  {≡
  }
```

Specialized Catch block

Example-3

```
try
{
  ≡
}
Catch(Exception e)
{
  ≡
}
```

Generalized Catch Blocks



Example - 4.

```
try
{
 Sop(10/0);
}
Catch(AE e)
{
 ==
}
Catch(Exception e)
{
 ==
}
```

Example - 5

```
try
{
 Sop(10/0);
}
Catch(Exception e)
{
 ==
}
Catch(AE e)
{
 ==
}
```

→ Unreachable Code. CTE.

Note:

Generalized Catch blocks & Specialized catch blocks can be written together but order should be first Specialized and then generalized.

Multiple try Catch Blocks:

If we want to handle more than one exception then we should go for multiple try catch blocks.

Example:

```
public class ExceptionDemo3
{
  public static void main(String[] args)
  {
    Sop("main method Started");
    int x=10, y=0;
    int[] a = new int[5];
    try{
      Sop("Entering try block of AE");
      Sop(x/y); // exp occured
```

```
        Sopl("Exiting try block of AE");
    }
    Catch(Exception e)
    {
        Sop("Entering Catch block AIOFBE");
        Sop(e.getMessage());
        Sop("Exiting Catch block AIOFBE");
    }
    try {
        Sop("Entering try block of AIOFBE");
        a[5]=100; //exp occurs.
        Sop("Exiting try block of AIOFBE");
    }
    Catch(Exception e)
    {
        Sop("Entering Catch block AIOFBE");
        Sop(e.getMessage());
        Sop("Exiting Catch block AIOFBE");
    }
    Sop("mme");
  }
}
```

1)
```
try
{
  ≡
}
Catch(_)
{
  ≡
}
```
✓

2)
```
try{
  ≡
}
Catch(_)
{
  ≡
}
Catch(_)
{
  ≡
}
Catch(_)
{
  ≡
}
```
✓

3)
```
try
{
  ≡
}
Catch(_)
{
  ≡
  try
  ≡
}
Catch(_)
{
  ≡
}
```
✓

4) try { } ✗

5) Catch( ) { } ✗

6) try { } finally { } ✓

7) try { } { try { } Catch(_) { } Catch(_) { } } ✗

8) try { } Catch(_) { } finally { } ✓

9) try { try { } Catch(_) { } } Catch(_) { } ✓

10) try { } Catch(_) { } finally { } finally { } ✗

11) try { } ——— ✗ Not allowed
Catch(_) { } In b/w try..Catch Nthg is allowed. ✗

Throw keyword:

1. It is a keyword used to throw both checked & unchecked ~~Obj~~ exception Objects explicitly.
2. 'Throw' will throw only those Objects which has the properties of throwable class.

throw new AE();
throw new NullPointerException();
throw new SQLExp();
throw new IOExp();
throw new CCE();
→ They Contain properties of Throwable Class.

throw new AgeInvalidExp();
throw new NotEligibleForMarriageExp();
throw new NotEligibleForJob();

→ Valid iff they have the Properties of throwable Class

3. Using throw we can throw only one exception Object at a time.

throw new AE(); ✓
throw new AE(), new NPE(); ✗

Exam

4. Throw must be used inside the method definition.

Examples

```
Public class ExceptionDemo1
{
  Sop ("main method Started");
  try
  {
    throw new ArithmeticException ("/by Zero");
  }
  Catch (ArithmeticException e)
  {
    Sop (e.getMessage());
  }
  Sop ("main method ended");
}
}
```

09/09/2018
SUNDAY

Ex: 1)

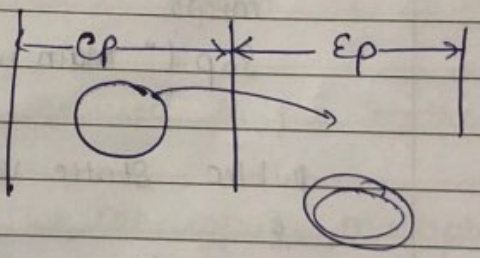
```
try
{ Sop ("Hello");
  Sop (10/0);
  Sop ("world"); X
}
Catch (AE e)
{
  ==
}
```

This line will not be executed. Decision will be made during Runtime

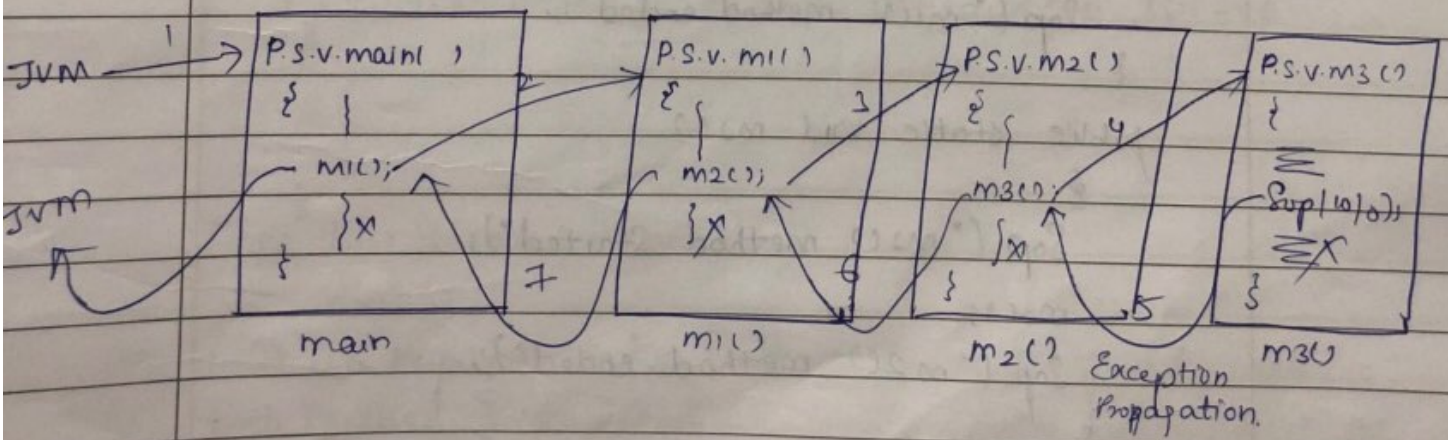
2)

```
try
{
  Sop ("Hello");
  throw new AE ("/by Zero");
  Sop ("world"); X CTE
}
Catch (AE e)
{
  ==
}
```

This line will not be executed. Decision will be made during Compile time



Exception Propagation:



1. When exception occurs in any method, and if that exception is not handled by the user that exception object is will be automatically propagated back to the Caller.

This propagation continues till the Jvm.

2. Exception object will be automatically propagated if it is a unchecked exception. If the exception object is checked we should inform the caller by using 'throws' keyword. This is because checked exceptions don't have the capability of propagating to the caller by itself.

Example

```
package ban.exceptions;
public class ExceptionDemo
{
  public static void main(String[] args)
  {
    Sop("Main method started");
    m1();
    Sop("Main method ended");
  }
  public static void m1()
  {
    Sop("m1() method started");
    m2();
    Sop("m1() method ended");
  }
  public static void m2()
  {
    Sop("m2() method started");
    m3();
    Sop("m2() method ended");
  }
  public static void m3()
  {
    Sop("m3() method started");
```

```
try
{
    Sop(10/0);
}
Catch (Exception e)
{
    Sop (e.getMessage());  // PrintStackTrace();
}                                  ↓
Sop ("m3() method ended");   used to know full details.
}                            where the exception is
}                            occuring.
```

'throws' Keyword:

1. It is a keyword used to inform the caller about the checked exception.
2. 'throws' keyword should be used in the method declaration.
3. Using 'throws' keyword we can inform multiple checked exceptions object to the caller.

Eg:

```
P.S.V.m1() throws SQLExp, InterruptExp, EOFExp
{
    {
    }
}
```

Diff b/w throw & throws:

throw	throws
1. It is a keyword used to throw check both checked & unchecked exception objects.	1. It is a keyword used to inform the caller about the checked exception.
2. It must be used in the method definition.	2. It must be used in the method declaration.

11/09/2018
Tuesday

3. We can throw only one exception 3. We can inform more than Object at a time. One checked exceptions objects to the caller.

Example-1:

```
Public class ExceptionDemo3
{
 Public static void main(String[] args) throws EOFException
 {
  Sop("Main method started");
  m1();
  Sop("Main method ended");
 }
Public static void m1() throws EOFException
 {
  Sop("m1() method started");
  m2();
  Sop("m1() method ended");
 }
public static void m2() throws EOFException
 {
  Sop("m2() method Started");
  m3();
  Sop("m2() method ended");
 }
Public static void m2() throws EOFException
 {
  Sop("m2() method Started");
  throw new EOFException();
 }
}
```

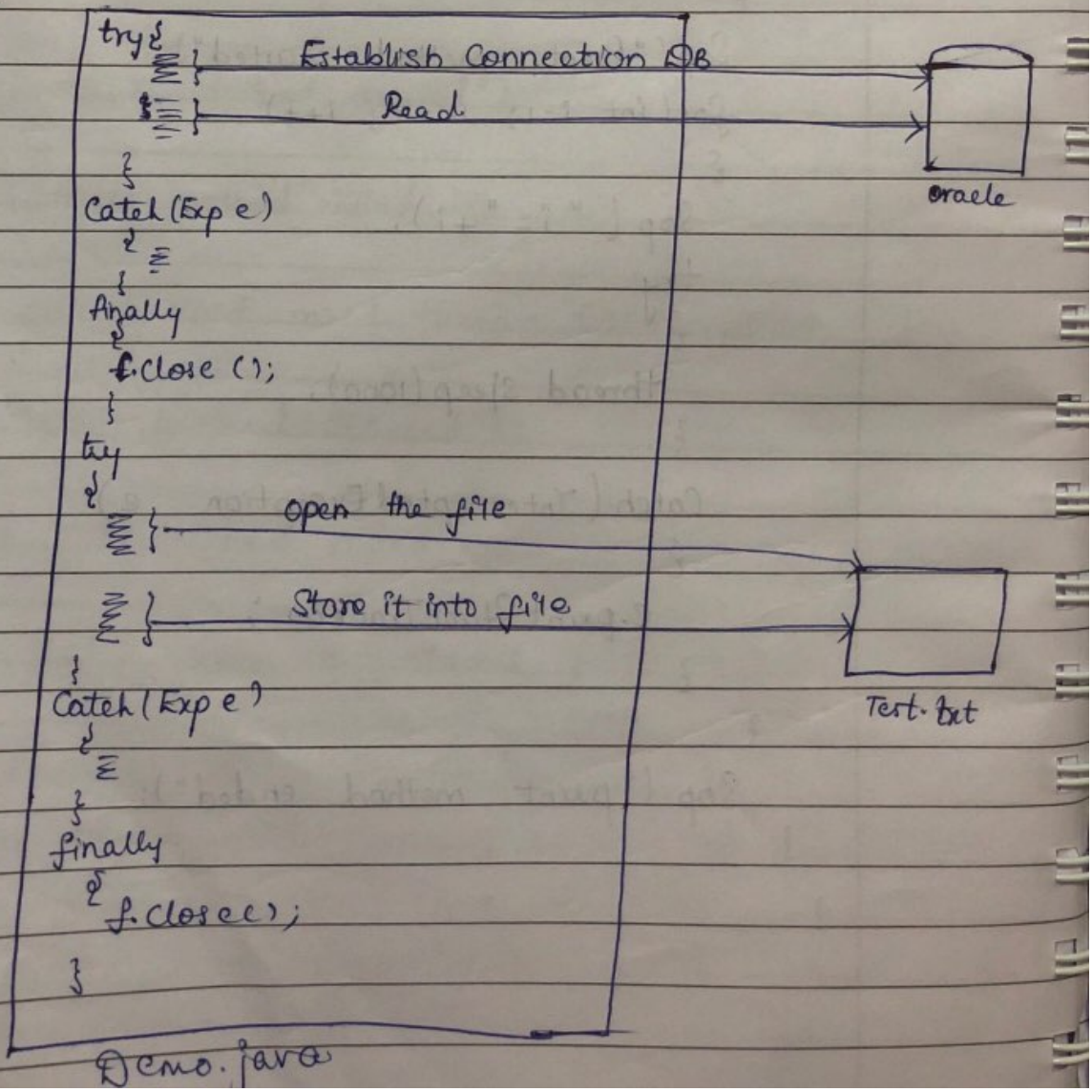
Example-2

```
Public class ExceptionDemo4
{
  Public Static void main(String[] args)
  {
    Sop("Main method started");
    PrintNumbers pn = new PrintNumbers();
    pn.print();
    Sop("Main method ended");
  }
}
public class PrintNumbers
{
  public void print()
  {
    Sop("Print method Started");
    for(int i=1; i<=10; i++)
    {
      Sop("i=" +i);
      try
      {
        thread.sleep(1000);
      }
      Catch(InterruptedException e)
      {
        e.printStackTrace();
      }
    }
    Sop("print method ended");
  }
}
```

14/09/2018
Friday

Checked Exception	Unchecked Exception
1. Known at Compile time but occurs at runtime	1. Known and occurs both at runtime
2 They don't have the capability of propagating the caller by itself.	2 They have the capability of propagating the caller by itself.
3. 'throws' keyword is required.	3. 'throws' keyword is not required.
4. Compiler forces to provide an alternative during Compilation time	4. Compiler doesn't force.

Finally block:



1. 'Finally block' usually contains closing related operations like closing the open database connection, closing the open file, closing the open session (or) terminal etc.,
2. 'Finally block' will always be executed irrespective of the exception.

1)
```
try
{
  Sop(10/2);   |✓
}
Catch(Exp e)
{
  ≡≡   |X
}
finally
{
  ≡≡   |✓
}
```

2)
```
try
{
  Sop(10/0);   |✓
}
Catch(Exp e)
{
  ≡≡   |✓
}
finally
{
  ≡≡   |✓
}
```

3)
```
try
{
  Sop(10/0);   |✓
}
finally
{
  ≡≡   |✓
}
```
JVM handles the Exception.

4)
```
try
{
  ≡≡
  return;   |✓
}
finally
{
  ≡≡   |✓
}
```

5)
```
try
{
  System.exit(0);   |✓
}
finally
{
  ≡≡   |X
}
```

6) try
{
new ArithmeticException(); | ✓ If we are not throwing the Catch block Can't be able to Catch the object.
}
Catch(Exception e)
{
Sop("Catch block executed"); | X
}

Custom Exceptions:

InsufficientBalaneeExeception.java:

```
paenage ban.exception;
public class InsufficientBalaneeException extends RuntimeException
{
  private String msg;
  public InsufficientBalaneeException(String msg)
  {
    this.msg = msg;
  }
  @Override
  public String getMessage()
  {
    return msg;
  }
}
```

BankTransaction.java:

```
public class BankTransaction
{
  public void validateBalaneee(double amtBal, double amtWithDraw)
  {
    Sop("ValidateBalaneee Method Started");
```

```
        if (amtBal > amtWithDraw)
        {
        Sop ("Succesful withDraw");
        amtBal = amtBal - amtWithDraw;
        }
        else
        {
        try
        {
        throw new InsufficientBalanceException
                                ("Maintain Sufficient Balance");
        }
      Catch (InSufficientBalanceException e)
        {
          Sop(e.getMessage());
        }
      }
      Sop ("Balance = " + amtBal);
      Sop ("ValidateBalance method ended");
  }
}
```

BankApp.java:

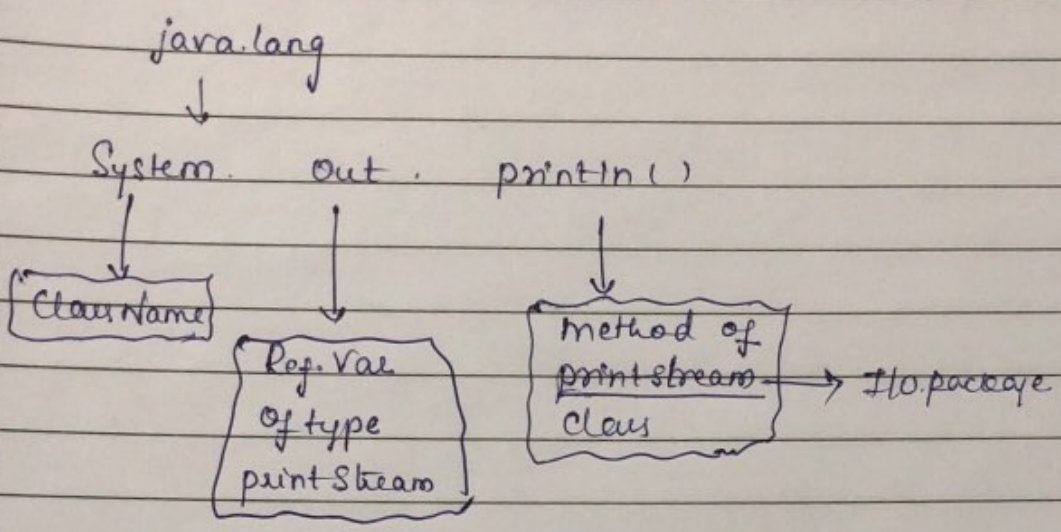
```
public class BankApp
 {
   public static void main (String[] args)
     {
       Sop ("Main method Staited");
       BankTransaction bt = new BankTransaction ();
       bt.validateBalance (50,000.00, 60000.00);
       Sop ("Main method ended");
     }
}
```

__/__/____

1. Create a Custom checked exception called notEligibleforMarriageException. Throw this exception when any boy whose age is less than 21 trys to marry or any girl whose age is less than 18 trys to marry.

2. Create a Custom exception called notEligibleForInterviewExp. Throw this exception when any student tries to attend an interview whose mock is not Completed.

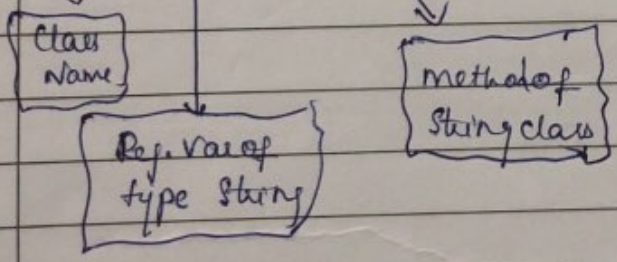
15/09/2018
Saturday



Ex:

```
class Test
{
    static String S = "Hello";
}
Test.S.length();
```

must be static



Private Constructor

1. When a class as a constructor the accessibility of a constructor (or) access modifier of the constructor is always equal to accessibility of the class unless being changed.
2. If the Constructor is given with access modifier as private then triggering such constructor Outside the class is not possible.
3. Which means Creation of an Object of a private Constructor class has to be done only with in the class respectively.
4. To access such class objects Out side the class we use helper methods and such helpers methods are public and static.
5. These helper methods are designed to get the objects of such private constructor class

Private_Const.java

```
package jsp.privateConst;
public class private_Const
{
  private Private_Const()
  {
    System.out.println("Running private Constructor body");
  }
  public Static Private_Const getInstance()
  {
    return new Private_Const();
  }
}
```

Main_Private_Const.java

```
package jsp.PrivateConst;
public Class Main_Private_Const
{
  public Static void main (String[] args)
```

```
}
    Private_Const P1 = Private_Const.getInstance();
        "       "  P2 =  "       "       "
    Sop("P1==P2" + (P1==P2));
}
}
```

Singleton-Classes

1. A class whose object can be created only once throughout the life of the project.
2. This can be achieved by using private constructor, helper methods and private data members.

Singleton_Class.Java:

```
Package jsp.PrivateConst;
public class Singleton_Class
{
    private static int count=0;
    private static Singleton_Class s=null;
    private Singleton_Class();
    {
      Count++;
      Sop(" Running Constructor body");
      Sop(" Count = "+Count);
    }
public static Sngleton_class getInstance()
    {
      if (Count==0)
        {
          S=new Singleton_Class();
        }
      return S;
    }
}
```

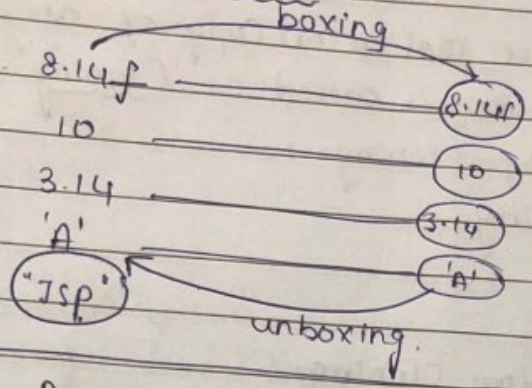
__/__/__

Singleton_class ~~for~~ Main.java.

```
P. S. v. m ( String[] args)
  {
   Singleton_class  S1 = Singleton_Class. getInstance();
   Singleton_class  S2 = Singleton_class. getInstance();
   Singleton_class  S3 = Singleton_class. getInstance();

   Sop ("(S1==S2) = " + (S1==S2)); // true
   Sop ("S1==S3) = " + (S1==S3)); // true
  }
}
```

15/09/2018
Saturday.

WRAPPER CLASSES:



Converting from primitive to Object is boxing

Vice versa is unboxing.

Primitive data type	Wrapperclass.
byte	Byte.java
Short	Short.java
int	Integer.java
long	Long.java
float	Float.java
double	Double.java
Char	Char.java
boolean	Boolean.java

* java.lang.
* final
* Comparable(I)
* (i) tostring()
  (ii) hashcode()
  (iii) equals(Object obj)
  → Overridden.

```
Test t = 10;
int x = new Test();  | X
```

```
Integer itr1 = 10; // Auto Boxing
Integer itr2 = Integer.valueOf(100); // manual Boxing.
                    ⇓
                  Static
```

itr1 → 10

itr2 → 100

```
int x = itr1; // Auto unboxing
int y = itr2.intValue(); // Manual Unboxing
                ⇓
            Non-static
Sop(x); // 10
Sop(y); // 100
Sop(itr1); // 10
Sop(itr2); // 100
  ↳ toString()
```

17/09/2018
Monday.

Cloning:

Creating an exact replica (or) Copy of an object with the same properties is called as "Cloning".

There are two types of cloning

i) Shallow cloning

ii) Deep cloning.

Program:

Steps to achieve Shallow Cloning:

1. The class should implement clonable interface.
2. Should override the clone method of object class by increasing the visibility to public.
3. Access the clone method.

Student.java (Shallow)

```
package ban.cloning;
public class Student implements Cloneable
{
    int id;
    String name;
    double marks;
    public Student(int id, String name, double marks)
    {
        super();
        this.id = id;
        this.name = name;
        this.marks = marks;
    }
    @Override
    public String toString()
    {
    return "id=" +id + ", name = " +name + ", marks = " +marks;
    }
```

```
@Override
public Object clone() throws CloneNotSupportedException
 {
   return Super.clone()
 }
}
```

MainClass.java:

```
package ban.Cloning;
public class MainClass
{
  P.S.V.M(String[] args) throws CloneNotSupportedException
  {
   Student S1 = new Student(1, "Dinga", 80.5);
   Student S2 = (Student) S1.clone();
   Sop(S1);
   Sop(S2);
   Sop(S1.hashcode());
   Sop(S2.hashCode());
   S1.id = 100;
   Sop(S1);
   Sop(S2);
  }
}
```

Program 2 (Shallow Drawback)

Student.java:

```
package ban.Cloning;
public class Student implements Cloneable
{
  int id,
  String name;
  double marks;
  Address addr;
```

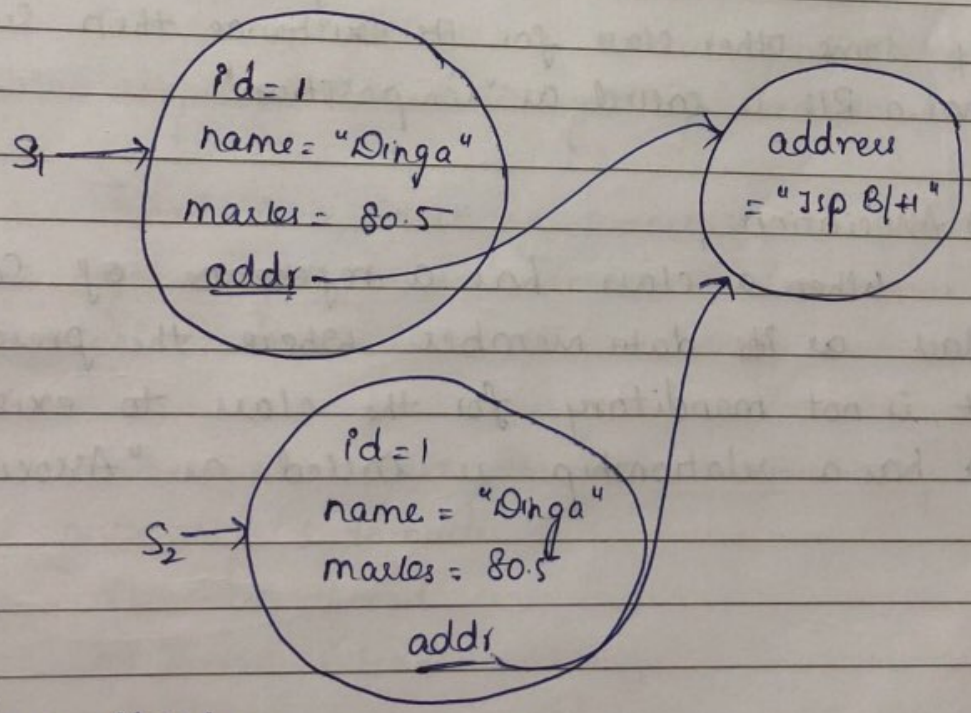
```
public Student(int id, String name, double marks, Address addr)
  {
   this.id = id;
   this.name = name;
   this.marks = marks;
   this.addr = addr;
}
@Override
 public String toString()
  {
  return "id=" + id + ", name=" + name + ", marks=" + marks +
                                    ", addr=" + addr;
}
@Override
public Object Clone() throws CloneNotSupportedException
   {
     return super.Clone();
   }
}
```

Address.java

```
package ban.Cloning;
public class Address
{
   String address;
   public Address(String address)
     {
       this.address = address;
     }
  @Override
    public String toString()
     {
       return "" + address;
     }
}
```

MainClass.java

```
P.S.V.M()
{
Address addr = new Address("Jsp Banaswadi/Hebbal");
Student S1 = new Student(1, "Dinga", 80.5, addr);
Student S2 = (Student) S1.clone();
Sop(S1);
Sop(S2);
Sop(S1.hashCode());
Sop(S2.hashCode());
Sop(S1.addr.hashCode());
Sop(S2.addr.hashCode());
S2.addr.address = "Jsp Basavanagudi";
Sop(S1);
Sop(S2);
}
```



Deep cloning.

Student.java

```
@Override
public Object Cloone() throws CloneNotSupportedException.
{
    Student S = (Student) Super.Clone();
```

```
S.addr = (Address) addr.clone();
return S;
}
```

HAS-A Relationship

When a class is having the reference of some other class as its data member then such type of relationship is called "HAS-A Relationship".

HAS-A R/s is of 2 types:

1) Composition.
2) Association.

1) Composition:

When a class manditoraly requires the reference of some other class for its existance then such a has-a R/s is called as "Composition".

2) Association:

When a class has a reference of some other class as its data member where the presence of it, is not manditory for the class to exists such a has a relationship is called as "Association".

MULTITHREADING:

18/09/2018
Tuesday.

Task:

Task is a set of instruction written by user to perform some activity.

Process:

A program in execution is known as a process. A process can have any number of threads (atleast one thread). If a process has single thread then we call that application as single-thread application If a process has multiple threads then we call such applications as multi-threaded application.

Number of processors in an application should be always less than (or) equal to number of threads.

When a process is executing which means internally a thread is getting executed.

Thread is a part of process hence we will call thread is a 'light-weight-process'. Thread is also known as "Execution Instance".

When we start the JVM, automatically it will start the following 3 threads.

i) Main thread
ii) Thread Schedular
iii) Garbage Collector.

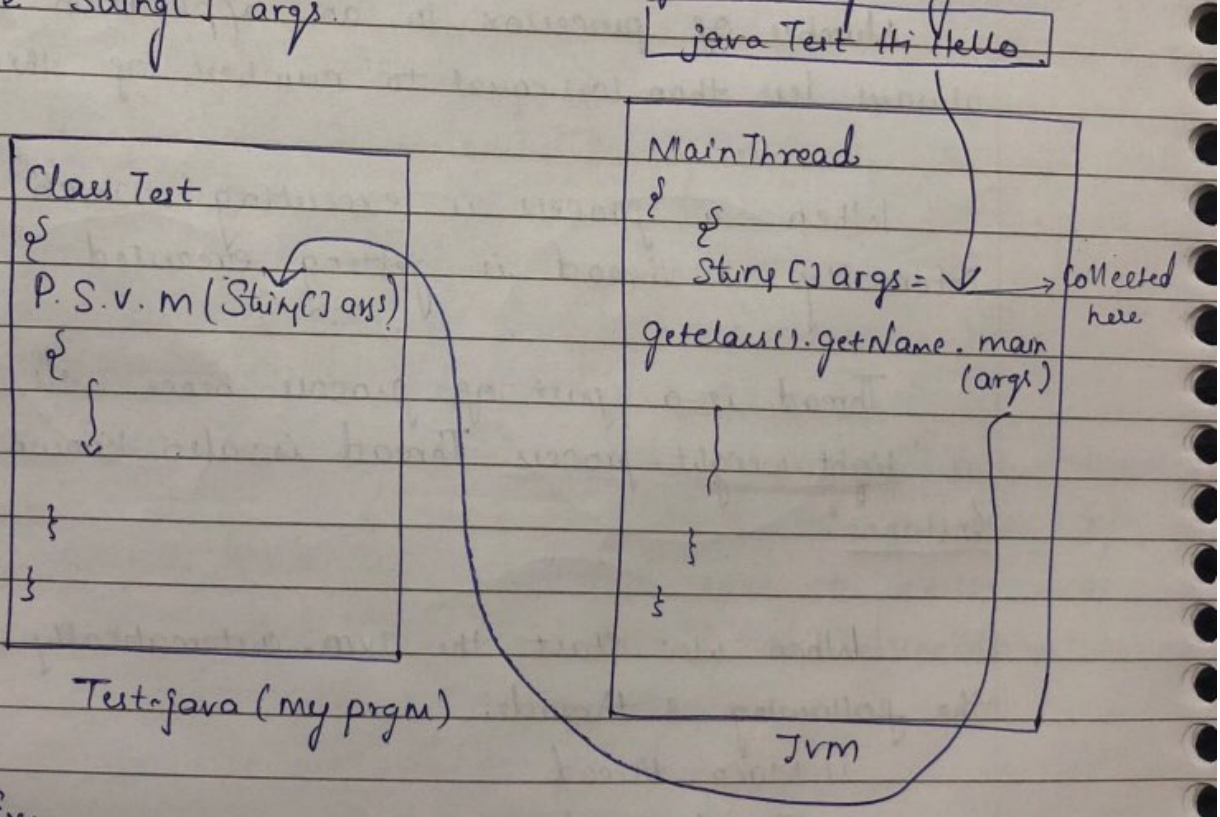
i) Main Thread:

It is responsible for execution of the program from the main method. Since main thread is a part of JVM which tries to call main method of our program by using the name of the class. Hence we write the main

method as Static. Main thread and our program are present in 2 different packages, if we want to access anything across the packages then they must be 'public'. Hence we declare main method as public.

Since main method doesn't return anything back to the main thread, hence return type of main method is void.

Main thread accepts array of String arguments (String[] args) from the Command prompt and pass it to main method. Hence main method accepts array of String arguments i.e String[] args.



Ex.

```
m1();
P.S. int m1()
{
    return 10;
}
```

In main method we can change return type because in main thread we are not collecting any thing so the program executes Normally

19/09/2018
Wednesday

ii) Thread Scheduler:

This thread is responsible for allocating the Separate System resources for the newly created threads. If we donot allocate the Separate System resources then all the multiple threads created by the user will not be able to work Concurrently.

Creating a thread:

We can create a thread by using one of the following 2 ways.

i) Extending thread class.
ii) Implementing runnable interface.

Thread Class:

→ It is available in java.lang package
→ It is used to create user defined threads.
→ It has overloaded constructors.
  * Thread t = new Thread();
  * Thread t = new Thread(Runnable r)
→ It has the following getters and Setters.
  * public String getName() } getters.
  * public long getId()
  * public int getPriority()
  * public void SetName(String name) } Setters.
  * public void setPriority(int priority)

Example:

```
package ban.threads;
public class ThreadDemo1
{
  P.S.V.M(String[] args) throws InterruptedException
  {
    Sop("Main method Started");
```

```
Thread t = Thread.currentThread();
for(int i=1; i<=10; i++)
 {
  Sop("Current Thread= "+t.getName());
  Sop(" i="+i);
  Thread.sleep(1000);
 }
 Sop("Original details");
 Sop("-------------");
 Sop("Priority= "+t.getPriority());
 Sop("Id= "+t.getId());
 t.setName("Ongo");
 t.setPriority(10);
 Sop("After Updating");
 Sop("-------------");
 Sop("priority= "+t.getPriority());
 Sop("Id= "+t.getId());
 Sop("Name= "+t.getName());
 Sop("Main method ended");
 }
}
```

Important Methods of Thread Class:

1) public static Thread currentThread():

This method returns an Object of current running thread. Once we get the Object of current running thread we can invoke getters and setters on it.

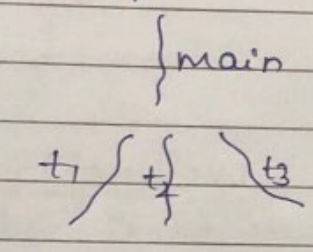
2) Public void start():

→ When we create a thread, that thread is said to be in born state. When the thread gets separate Systemresources like System memory and cpu time then thread will be in running state.

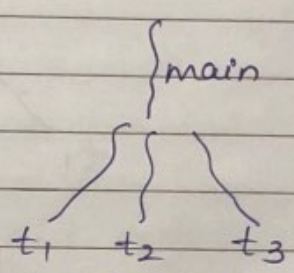
It is a responsibility of thread Scheduler to allocate Separate system resources for the newly created threads. Thread Scheduler will be called automatically when we invoke Start method.

3) Public void run():

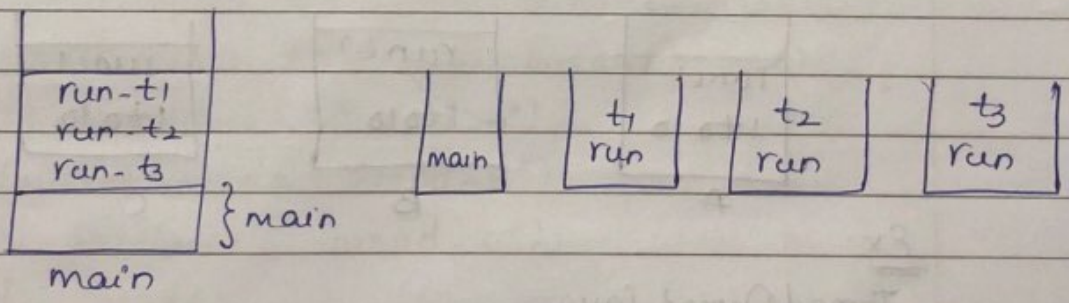
run() method defines the job of the thread. This run() method will be executed automatically when the thread gets its CPU time.



t1.run();
t2.run();
t3.run();



t1.Start();
t2.Start();
t3.Start();



Difference between Start() and run() method:

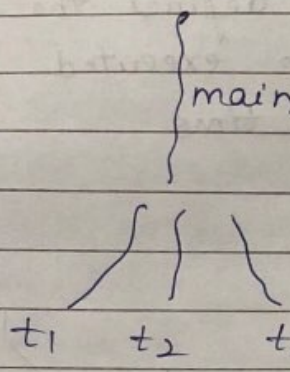
→ When we invoke Start() method thread Scheduler will allocate Separate system memory. When those threads gets its CPU time they will be start executing the run method of each thread concurrently.

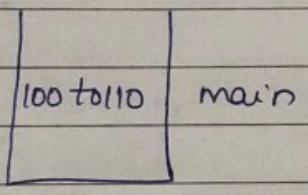
→ When we invoke run method no Separate System memory will be allocated, run() method will be executed just like any other method in the Same System memory.

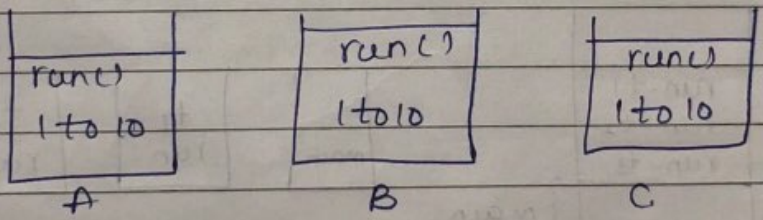
4) public void stop():

When we invoke this method on a running thread, the thread will reach to Dead State(Stopped State). Once the thread is stopped we can't start again.

Creating a thread by extending thread class.







Ex:

ThreadDemo3.java:

```
package ban.Threads;
public class ThreadDemo3 extends Thread
{
  @Override
  public void run()
  {
    for(int i=1; i<=10; i++)
    {
      Sop("Thread Name=" + Thread.CurrentThread().
                                    getName());
```

```
        Sop("i= "+i);
        try
        {
          Thread.sleep(1000);
        }
        Catch(InterruptedException e)
        {
          e.printStackTrace();
        }
      }
    }
  }
```

MainClass.java:

```
public static void main(String[] args)
{
  Sop("Main method started");
  ThreadDemo3 t1 = new ThreadDemo3();
  ThreadDemo3 t2 = new ThreadDemo3();
  ThreadDemo3 t3 = new ThreadDemo3();
  t1.setName("Thread-A");
  t2.setName("Thread-B");
  t3.setName("Thread-C");
  t1.start();
  t2.start();
  t3.start();
  for(int i=100; i<=110; i++)
  {
    Sop("Thread Name = "+Thread.currentThread().getName());
    Sop("i= "+i);
  }
  Sop("Main Method ended");
}
```

20/09/18
Thursday

In the above example, three userdefined threads were sharing same body.

Creating four threads with four different body:

```
package ban.threads;
public class addition extends thread
 {
  @Override
  public void run()
  {
  Sop("thread name = "+thread.Currentthread().getname());
  int x=10, y=20;
  Sop("Sum = "+(x+y));
  }
}
public class Substraction extends thread
  {
   @Override
   public void run()
   {
   Sop("thread name = " + thread.Currentthread().getname());
   int x=10, y=20;
   Sop("Sub = "+(x-y));
   }
}
public class multiplication extends thread
   {
    @Override
    public void run()
    {
    Sop("thread name = "+thread.CurrentThread().getname());
    int x=10, y=20;
```

```
        Sop("mul = "+(x*y));
    }
}
public class division extends Thread
{
@Override
public void run()
{
Sop("threadname = thread.Currentthread().getname());
int x=10, y=20;
Sop("div = "+(x/y));
}
}
public class MainClass
{
public static void main(String[] args)
{
Sop("thread name = "+thread.Currentthread().getName());
Addition a = new Addition();
Substraction S= new Substraction();
Multiplication m= new Multiplication();
Division d = new Division();
a.SetName("Addition Thread");
S.SetName("Substraction Thread");
m.SetName("Multiplication Thread");
a.Start();
S.Start();
m.Start();
d.start();
Sop("Main method ended");
}
}
```

Thread Synchronization

1. When a method is shared among multiple threads then all threads tries to share the same method and produces inconsistent result.
2. In Order to get the consistent result we need to Synchronize the threads.
3. When a method is shared among multiple threads such methods are said to be thread unsafe methods.
4. We can achieve thread Synchronization by using the keyword "Synchronized".
5. Internally Synchronization Concept is implemented by using the concept of "Locks".

→ Locks are of 2 types:
   i) Object level locks.
   ii) Class level locks.

6. Locks can be applied on Object level and class level but not on "Method level".

7. If any thread wants to execute any Synchronized method of one object it should be the owner of that object ("that thread should own the lock")ie. lock will be released only for that thread and all threads will be locked.

8. When one thread own the lock and starts executing Synchronized method then no other thread will be able to execute any Synchronized method of that object.

9. All the threads who want to execute the Synchronized method automatically they enter into wait state. Once the thread releases the lock then any of the waiting thread might own the lock.

10. Usually the methods which are responsible for performing update operation needs to be Synchronized.

11. If a method is Synchronized, such methods are said to be thread safe methods.

12 Thread Synchronization will produce the Consistent result but it will increase the waiting time of a thread. Hence, methods which are responsible for performing only read operation should not be Synchronized.

13. All thread unsafe methods will not cause a prblm only those methods which performs update operation will produce the inconsistent result.

Ex:- Synchronized public void amtwithdraw()
{
}→ Update → It Should be Synchronized.
}

Synchronized public void amtDeposit()
{
}→ Update
}

public void CheckBalance()
{
}→ read → need not be Synchronized.
{

21/09/2018
Friday.

Ex-2:

```
Synchronized public void BookTicket()
  {
     |→Update
  }
Synchronized public void cancelTicket()
  {
     |→Update
  }
public void getAvailableSeats()
  {
     |→Read.
  }
```

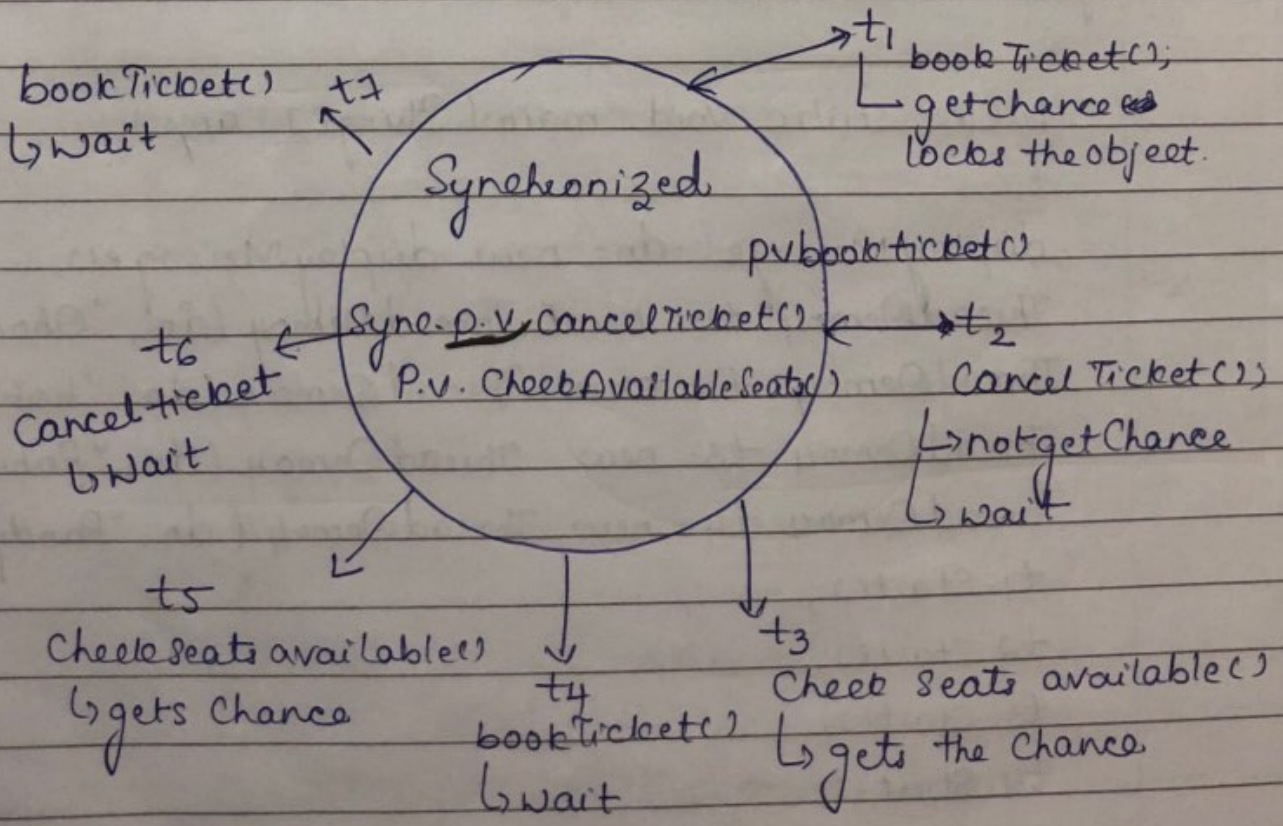
Example,

```
package ban.threads;
public class DisplayMessage
{
 Synchronized public void disp(String msg)
   {
     for( int i=1; i<=10; i++)
      {
       Sop(" Good Morning ->");
       Sop(msg);
       try
       {
       thread.sleep(1000msec);
      }
      Catch(InterruptedException e)
       {
        e.printStackTrace();
       }
   }
}
```

__/__/____

```
package bar.threads;
public class ThreadDemo4 extends Thread
{
  private displayMessage dm;
  private String msg;
  public ThreadDemo4(displayMessage dn, String msg)
    {
    this.dn= dn;
    this.msg = msg;
    }
  @Override
  public void run
    {
    dn.disp(msg);
    }
}
public class MainClass3
  {
  public static void main(String[] args)
    {
  displayMessage dn= new displayMessage();
  ThreadDemo4 t1= new ThreadDemo4(dn, "Dhoni");
  ThreadDemo4 t2= new ThreadDemo4(dn, "kohli");
  ThreadDemo4 t3= new ThreadDemo4(dn, "Rohith");
  ThreadDemo4 t4= new ThreadDemo4(dn, "Pandy");
  t1.start();
  t2.start();
  t3.start();
  t4.start();
  }
}
```

```
class ReservationSys
{
  Synchronized public void bookTicket()
  {
  }
  Synchronized public void cancelTicket()
  {
  }
  public void checkSeatAvailable
  {
  }
}
public static void main(String[] args)
{
ReservationSys R = new ReservationSys();
}
```



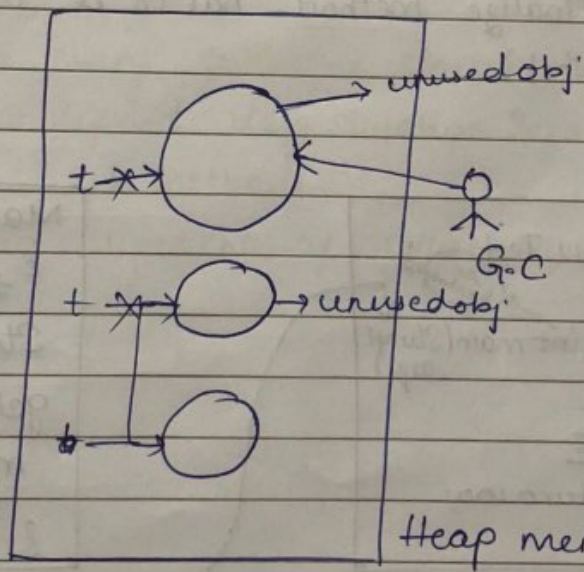
__/__/____

Garbage Collector:

1. It is a low priority thread started by Jvm whose responsibility is to clean up the unused objects from heap memory.
2. Just before deallocating the object by Garbage Collector, Garbage collector will automatically execute 'finalize' method of the respective object then deallocates the memory.
3. We can call the garbage collector explicitly by using System.gc(); (gc() is a static method of class System);

4. Making used objects as unused objects.

```
Test t = new Test();
t = null;
```



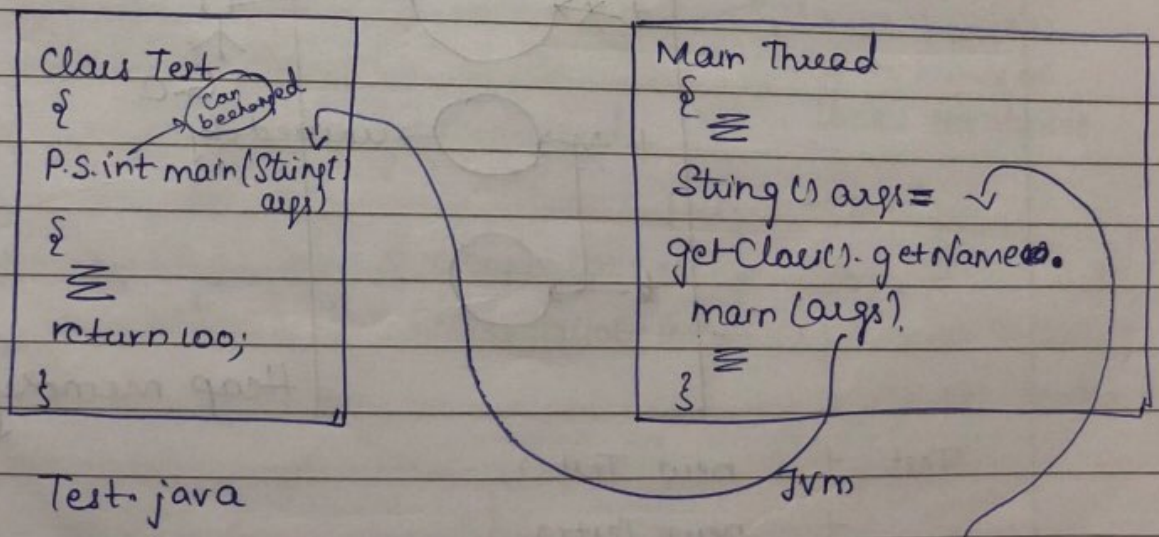

```
Test t = new Test();
t = new Test();
```

Ex: Public class MainClass

```
{
  public static void main(String[] args)
  {
    Test t = new Test();
    t = null;
```

```
        System.gc();
    }
}

public class Test
{
    @Override
    public void Finalize()
    {
    Sop("Current Thread = "+t.thread.CurrentThread().getname());
    Sop("Executed finalize method");
    }
}
```

Note: If we are deallocating the object of a class(Test) then finalize method has to be written in the same class (Test).



Compile javac Test.java
Run java Test → in Command prompt
Hi Hello.

26/09/2018
Wednesday

Collections

DrawBacks Of Arrays:

1. Arrays are fixed in Size.
2. Arrays stores only homogeneous data. But in real time all the data might not be homogeneous.
3. Arrays always demands for Consecutive memory locations hence it doesn't utilize the memory effectively.
4. No readymade methods are available. Because array doesn't implemented on any standard data Structure.

To overcome all the above drawbacks of arrays we will go for Collection.

Diff b/w Array and Collection.

Array	Collection
1. It is fixed in Size	1. Expandable or growable in nature.
2. Stores only homogeneous data.	2. Stores both homogeneous and heterogeneous data.
3. It always demands for Continuous memory location. Hence it does not utilize the memory effectively.	3. It doesn't demand for Continuous memory location. It utilizes the memory effectively Compared to arrays.
4. No readymade methods are available. Bcoz it doesn't implemented on any Standard data Structure	4. Many readymade methods are available. Bcoz every Collection is implemented on some Standard data Structure.
5. Performance is better Compared to Collection.	5. Memory utilization is better Compared to array.
6. We can Store both primitive and non primitive by declaring respective type of array.	6. We can Store only Objects.

__/__/____

Collection FrameWork Library

It is a library which contains set of classes and interfaces written by the java people to overcome the drawbacks of arrays. This library is present in java.util package.

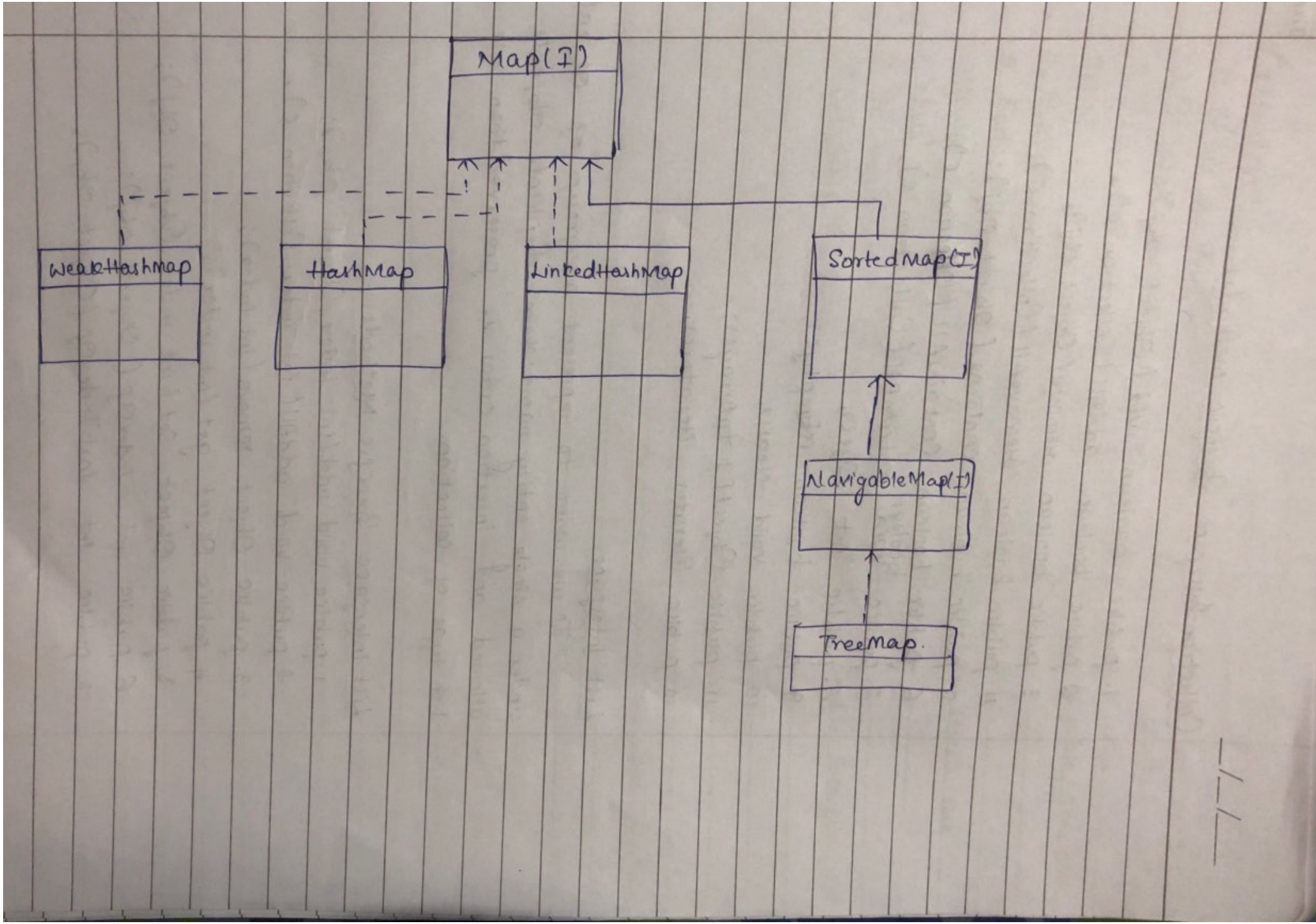
Java — C++

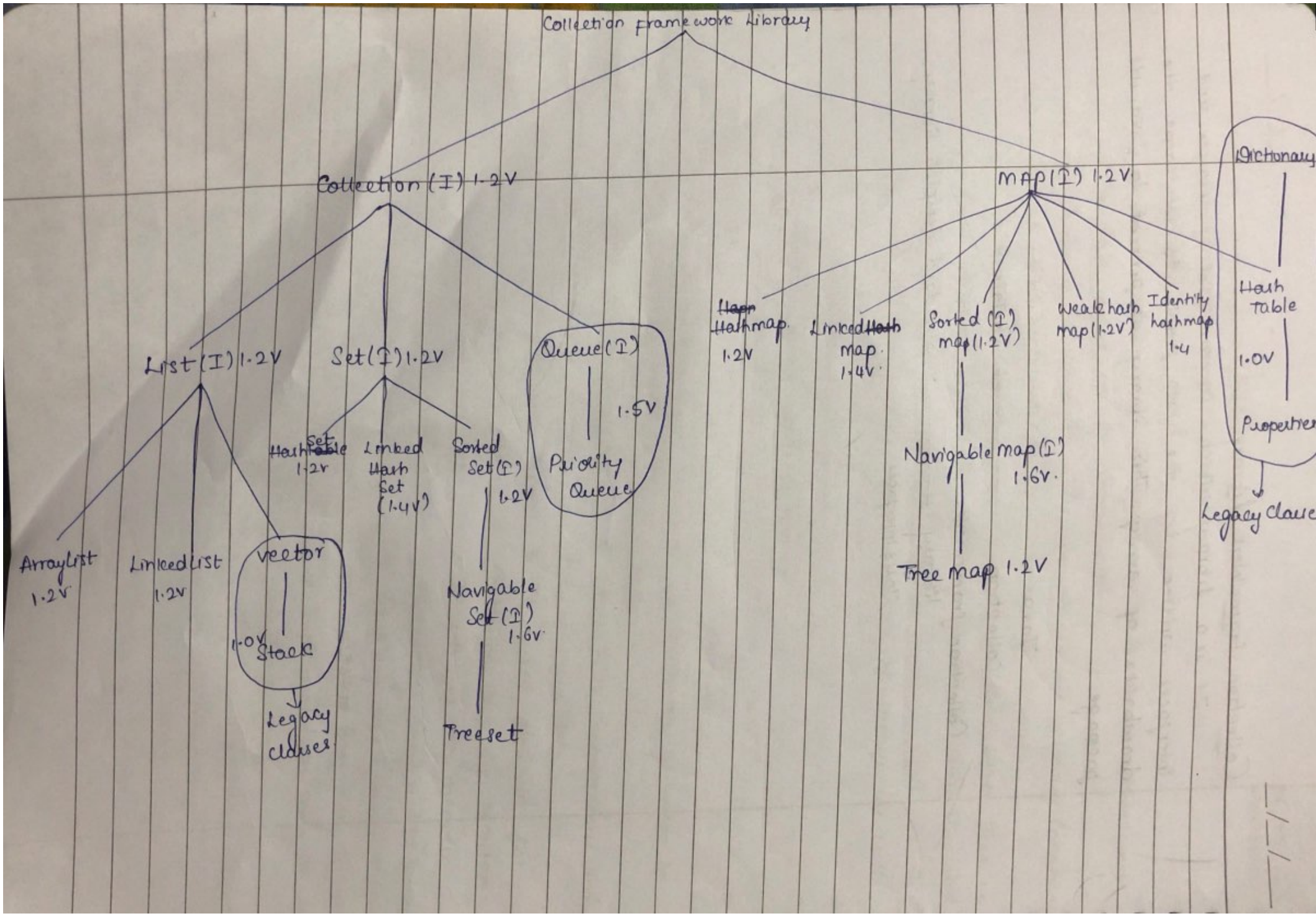
Collection ⟶ Container.

Collection framework library ⟶ Standard template library.

↓

Class & interfaces.





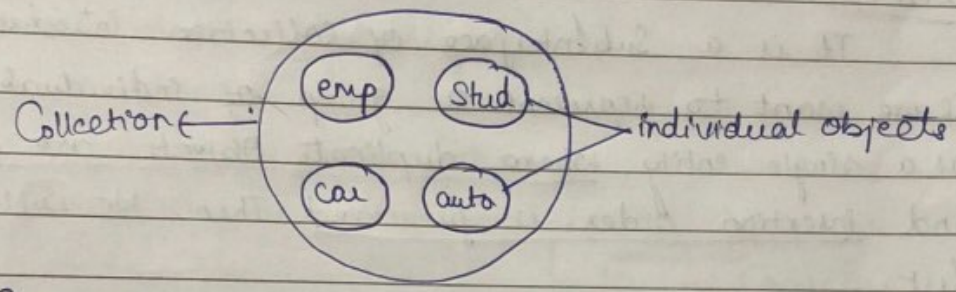
27/09/2018
Thursday

Collection (Interface):

It is a Supermost interface in entire collection frame-work library introduced in 1.2V. This interface doesn't have any class which directly implements. This interface contains the most commonly used methods which are applicable for the most of the Collection.

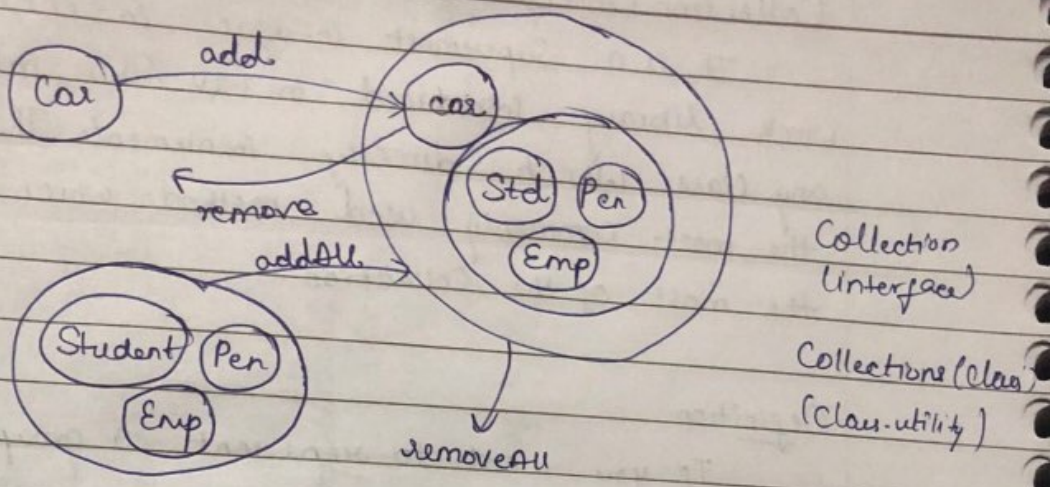
Definition:

If you want to represent a group of individual objects as a single entity then we will go for Collection.



Collection interface Methods:

1. public boolean add(Object obj)
2. public boolean remove(Object obj)
3. public boolean addAll(Collection c)
4. public boolean removeAll(Collection c)
5. public boolean contains(Object obj)
6. public boolean containsAll(Collection c)
7. public boolean retainAll(Collection c)
8. public void clear()
9. public int size()
10. public boolean isEmpty()
11. public Object[] toArray()
12. public Iterator iterator()



List (I):

It is a SubInterface of Collection introduced in 1.2V. If we want to represent a group of individual Object as a single entity, where duplicate Objects are allowed and insertion order is preserved Then, We will go for List.

List Interface Methods:

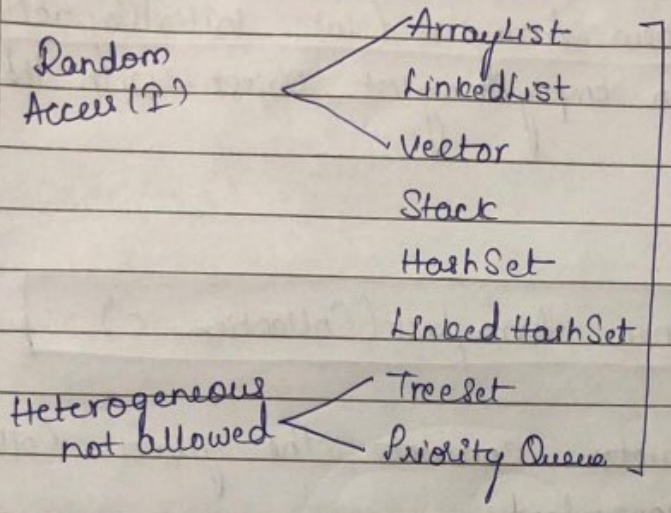
1. public void add(int index, Object obj)
2. public Object remove(int index)
3. public boolean addAll(int index, Collection c)
4. public int indexOf(Object obj)
5. public int lastIndexOf(Object obj)
6. public Object get(int index)
7. public Object Set(int index, Object obj) //replace
8. public ListIterator listIterator()

} index based method

ArrayList: (I):

It is an implementation class of List introduced in 1.2V. The underlined dataStructure is Resizable or growable array. Duplicate Objects are allowed and insertion order is preserved.

Heterogeneous objects are allowed. Null insertion is possible. It implements Serializable interface, Cloneable interface and random access interface.



Constructors of ArrayList

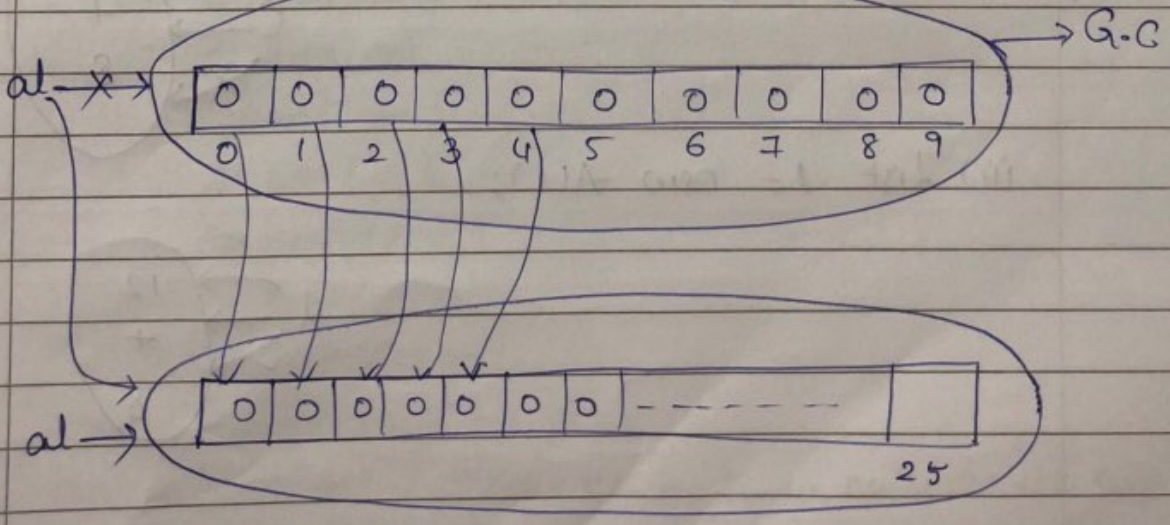
1. `ArrayList al = new ArrayList();`

∴ Default Capacity = 10 → Capacity

new Capacity = Current Capacity * 3/2 + 1

= 10 * 3/2 + 1 => 15 + 1

= 16



This Constructor will create an empty ArrayList object with the default initial capacity → 10.

Once, the arraylist is maximum Jvm will Create a new arraylist object as per the above mentioned formula.

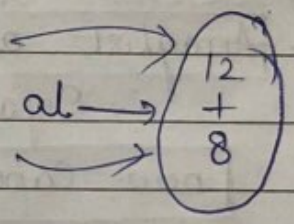
2. Arraylist al = new Arraylist (int initialCapacity).

Creates an EmptyArraylist Object with the Specified Initial Capacity.

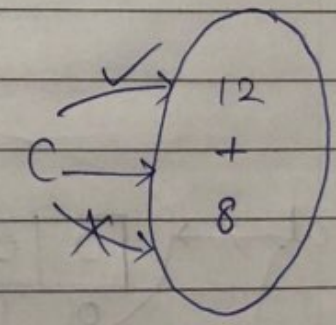
3. Arraylist al = new Arraylist(Collection c)

This constructor converts the given Collection into Arraylist equivalent.

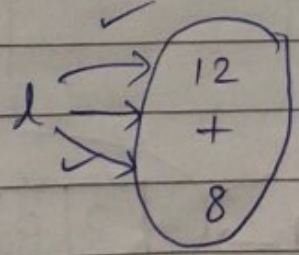
i) AL al = new AL();



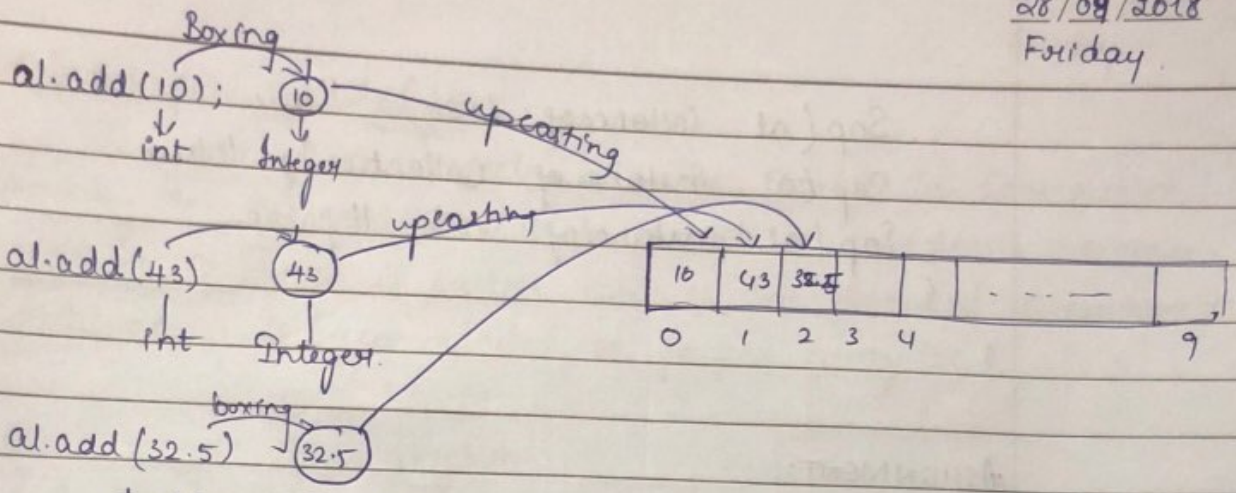
ii) Collection c = new AL();
List l = (list) c;



iii) List l = new AL();



28/09/2018
Friday



Example:

```
package awt.cfl;
public class ArrayListDemo1.java
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        Sop("Before = "+al.size()); //0
        al.add(10);
        al.add(43);
        al.add(32.5);
        al.add(10);
        al.add(null);
        al.add('A');
        al.add("Jsp");
        al.add("Osp");
        al.add(656.64);
        al.add(true);
        Sop(al); //[10, 43, 32.5, 10, null, A, Jsp, Osp, 656.64, true]
        Sop("After = "+al.size());
        Sop(al instanceof Cloneable); //true
        Sop(al instanceof Serializable); //true
        Sop(al instanceof RandomAccess); //true
```

```
Sop(al instanceof List); //true
Sop(al instanceof Collection); //true
Sop(al instanceof Set); //false.
}
}
```

ASSIGNMENT:

1. In an arraylist of heterogeneous Objects
   a) WAP to display only integer Objects.
   b) WAP to display only String Objects.
   c) WAP to display only even integer Objects.
   d) WAP to display both integer and double Object.
   e) WAP to display all the String Objects starts with 'S'.
   f) WAP to display all the String Objects which contains 'a' in it.
   g) WAP to remove all the Objects except integer and String.

02/10/2018
Tuesday

Advantages of ArrayList:

In arrayList objects will be stored in Consequtive memory location and since it implements random access interface Retival Operation is faster. Hence if our frequent operation is retival it is recommanded to go for arraylist.

Disadvantages of ArrayList

1. In arraylist all objects will be stored in Consequtive memory location, any insertion or deletion in between internally involves several shift operations, which might brings down the performance of the application. Hence, if our frequent Operation is insertion (or) deletion in between, then arrayList is not recommanded.

NOTE: Diff blw Array & ArrayList is same as array & Collection

2. Linked List:

1. It is an implementation class of list introduced in 1.2V
2. The underlined dataStructure is doubly linked list.
3. Insertion Order is preserved, duplicates are allowed, null insertion is possible, heterogeneous objects are allowed, it implements serializable interface, Cloneable interface but not randomAccess.

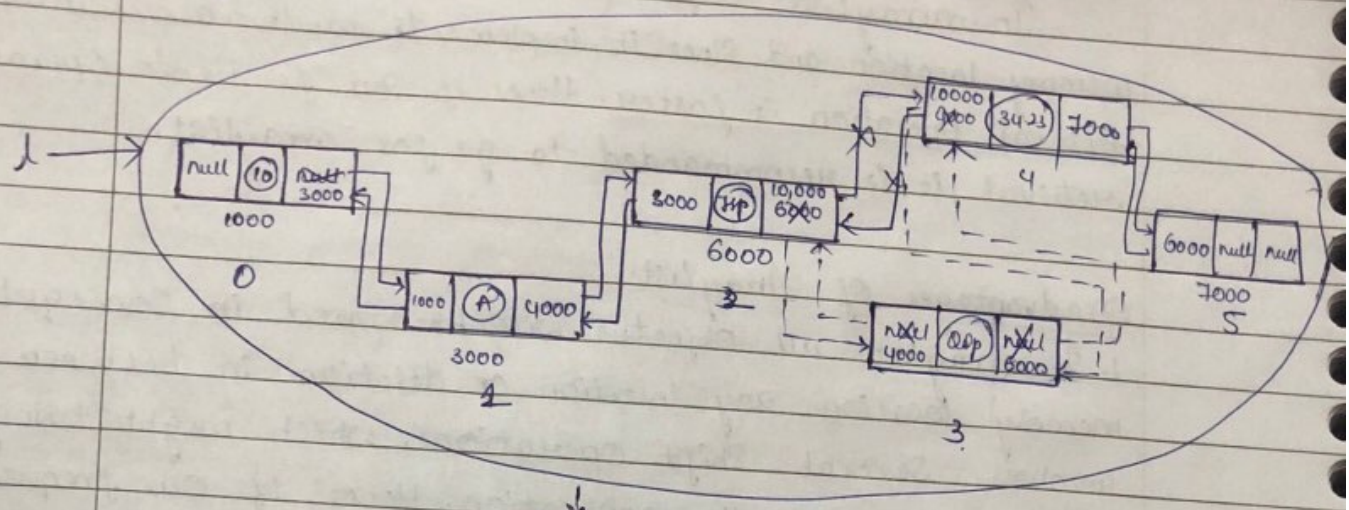
Constructors of Linked List:

1. LinkedList l = new LinkedList();
2. LinkedList l = new LinkedList (Collection C);

Methods of LinkedList:

1. addFirst(Object Obj)
2. addLast(Object obj)
3. removeFirst();
4. removeLast();

5. getFirst();
6. getLast();



↓
LinkedList.

Advantages of LinkedList:

1. Since Objects will not be stored in Consequtive memory location, any insertion (or) deletion in b/w the linkedlist doesn't involve any Shift Operation. Hence insertion (or) deletion in between the linkedlist is faster.

Disadvantage of LinkedList:

Since it doesn't Implement randomaccess, Objects will not be in Consequtive memory location, retrieval Operation is slower. Hence if our frequent Operation is retrieval, Linked list is not recommanded.

Diff b/w ArrayList & LinkedList

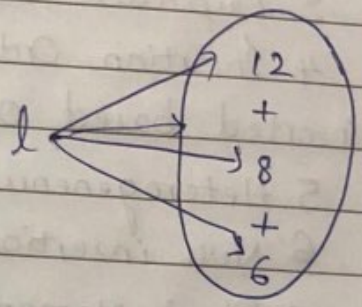
ArrayList	LinkedList
1. Underlined datastructure is growable array.	1. Underlined DS is doubly linked list
2. It implements Randomaccess	2. Doesn't implements randomAccess

03/10/2018
Wednesday

3. Objects will be Stored in Consequtive memory location.
4. Retrieval operator is faster.

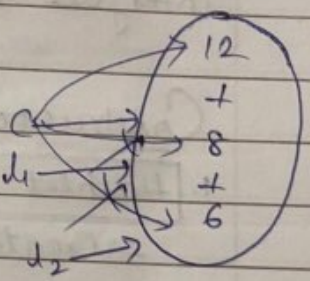
3. Objects will not be Stored in Consequtive memory location.
4. Insertion & deletion in between linkedlist is faster.

1) LinkedList l = new LinkedList();



2) Collection c = new LinkedList();

List l_1 = (List) c;
LinkedList l_2 = (LinkedList) l;



Set:

1. It is a Sub-interface of Collection introduced in 1.2V.
2. If we want to represent a group of individual objects as a single entity where duplicate objects are not allowed and insertion order is not preserved, then we will go for Set.

Diff blw List & Set

List	Set
1. Duplicate objects are allowed	1. Duplicate objects are not allowed.
2. Insertion order is preserved.	2. Insertion order is not preserved.

HASH SET:

1. It is an implementation class of Set introduced in 1.2V.
2. The underlined datastructure is Hashtable
3. Duplicate Objects are not allowed.
4. Insertion Order is not preserved. All objects will be inserted based on the hashcode.
5. Heterogeneous Objects are allowed
6. Null insertion is possible.
7. It implements Serializable interface, Cloneable interface butnot randomaccess.

Constructor of Hashset:

1. `Hashset h = new Hashset()`

→ Creates an empty hashset object with the default initial capacity 16 and default fill ratio = 0.75.

→ When the Hashset reaches 75% of the given initial capacity then it will create a new hashset Object.

2. `Hashset h = new Hashset (int initialCapacity)`

→ Creates an empty hashset Object with the Specified initial capacity.

3. `Hashset h = new Hashset (int initialCapacity, float fr)`

Creates an empty hashset Object with the Specified initial capacity and Specified filled ratio.

4. `HashSet h = new Hashset (Collection c)`

Converts the given Collection into hashset equivalent.

Example for Hashset:

```
public class HashSetDemo1
{
    public static void main(String[] args)
    {
        HashSet h = new HashSet();
        h.add(10);
        h.add(4.34);
        h.add(46.32);
        h.add('A');
        h.add("JSP");
        h.add(null);
        h.add("OSP");
        h.add(54);
        h.add(234);
        h.add(21);
        h.add(21);
        Sop(h); // Insertion Order is not preserved.
        Sop(h instanceOf Serializable); //true
        Sop(h instanceof Cloneable); //true
        Sop(h instanceOf RandomAccess); //false
    }
}
```

Imp Cursors in Collection Framework library.

- Java Collection framework library provides the following 3 types of cursors which are used to retrieve the objects from the collection one by one.

1. Enumeration (I):

Only used in legacy classes, Legacy Cursor, 1.0v, Only read method and Contains 2 Methods.

2. Iterator (I):

→ It is a universal Cursor introduced in 1.2V.
→ It performs read and remove Operations.
→ Totally 3 methods are present.
→ It is unidirectional (Forward).

3. ListIterator (I):

→ ListIterator is a powerful Cursor.
→ It is Bidirectional.
→ It is a Sub-interface of iterator (I).
→ It performs read, remove & replace Operation.
→ Totally 9 methods are present.
→ Applicable only for list implementation classes.

Helper Methods:

CollectionFrameWork library provides the following 2 helper methods which helps us in providing the implementation class objects of respective Cursors.

1) public Iterator iterator(): → Collection (I)

→ This helper method is available in Collection interface.

→ This helper method returns (or) provides an implementation class Object of iterator interface.

→ This method is implemented in all the Collection

Eg:

```
interface Iterator
{
    boolean hasNext();
    Object next();
    void remove();
}
```

Iterator itr = C.iterator(); → Provides implementation class Object.

↓ Interface type reference.

itr.hasNext();
itr.next();
itr.remove();

- ArrayList
- LinkedList
- Vector
- Stack
- HashSet
- LinkedHashSet
- TreeSet
- PriorityQueue

2. Public ListIterator listIterator() → List(I)

→ This method is available in list interface.
→ This method provides implementation class Object of list iterator Cursor interface.
→ This method is implemented only in the list implementation classes.

```
Interface ListIterator extends Iterator
{
    boolean hasNext()
    Object  next()          } Forward.
    int     nextIndex()

    boolean hasPrevious()
    Object  previous()      } Backward.
    int     previousIndex()

    Void remove()
    Object set(Object Obj)
    void   add(Object obj)
}
```

Inherited from Iterator(I)

ListIterator itr = C.listIterator(); Provides implementation class Object of listIterator(I)

interface Type Reference.

→ ArrayList
→ LinkedList
→ Vector
→ Stack

Example:

Retrieve Only Integer Objects:

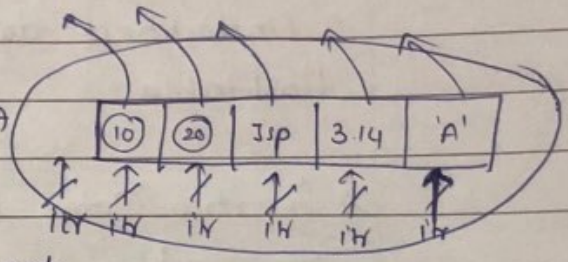
```
P.S.V.M(String[] args)
 {
   HashSet h = new HashSet();
   h.add(10);
   h.add(20);
   h.add('A');
   h.add('Jsp');
   h.add(54);
   h.add(234);

  Iterator itr = h.iterator();

  While(itr.hasNext())
   {
     Object obj = itr.next();
     if(obj instanceOf Integer)
       {
        Sop(obj); // toString.
       }
   }
}
```

Olp:
10
20
54
234.

Working Of Iterator Cursor:

```
Iterator itr = al.iterator();
While (itr.hasNext())
{
    itr.remove();
}
```

Exp: IllegalStateException




```
Iterator itr = al.iterator()
While (itr.hasNext())
{
    Object obj = itr.next();
    itr.remove();
}
```

→ Points to the next Object. ✓

If we write itr.hasNext(), it will check whether the next object is present (or) not. Initially, itr will points before the first object after giving the itr.next(); Only, itr will points to object and then it will remove it.

Linked Hashset:

It is exactly same as hashset except the following:

Diff b/w List and Set

List	Set
1. Duplicated elements are allowed.	1. Not allowed.
2. insertion order is presened	2. Not presened

If we want insertion order and Duplicates are not allowed we will go for linkedhashSet.

Hashset	LinkedHashSet
1. Underlined datastructure is Hashtable	1. Underlined datastructure is Hashtable and doubly linked list
2 Insertion Order is not preserved.	2 Insertion Order is preserved.

Example,

```
import java.util.LinkedHashSet;
public class LinkedHashSet1
  {
    P.S.V.M (String[] args)
    {
    LinkedHashSet h = new LinkedHashSet();
    h.add(10);
    h.add(10);
    h.add(20);
    h.add("Jsp"); Sop(h);
  }
}
```

Olp:

10
20
Jsp.

} Insertion Order is preserved & Duplicates are not allowed.

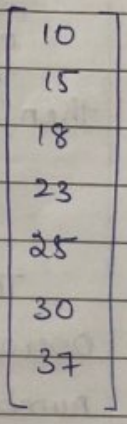
04/11/2018
Thursday.

SortedSet(I):

1. It is a Sub-interface of Set introduced in 1.2V. If we want to represent a group of individual objects as a single entity where objects needs to be sorted then we will go for SortedSet.
2. Sorting can be either ascending order (or) Descending order.
3. Ascending Order Sorting is known as "Default Natural Sorting Order (DNSO)" where as Descending Order Sorting is known as "Customized Sorting Order (CSO)".

SortedSet Methods:

1. first(); //10
2. last(); //37
3. headset(25); [10,15,18,23]
4. tailset(23); [23,25,30,37]
5. Subset(15,30); [15,18,23,25].



Treeset:

1. It is an implementation class of Navigable Set introduced in 1.2V.
2. The underlined data structure is Balanced Tree (BST).
3. Duplicates are not allowed.
4. Insertion order is not preserved. All objects will be inserted either in default natural sorting

order or Customised Sorting order.

5. Heteregeneous Objects are not allowed.
6. Null insertion is not possible.
7. It implements Serializable interface, Cloneable interface but not random access.

Into the TreeSet Collection to add Objects the following two prequisites / Conditions needs to be Satisfied.

1. Object must be homogeneous type.
2. Object must implement either Comparable interface (or) Comparator interface.

If any one of the above Conditions are not Satisfied then we will get class cast exception.

10	5	8	15	12

(2)

To Compare two objects we will go for relational operator. But in Treeset we use to Subtract the numbers. The possibilities if we will get if we Subtract

10-5
- +ve ($n_1 > n_2$)
- -ve ($n_1 < n_2$)
- 0 ($n_1 == n_2$)

Comparable Interface:

1. It is a functional interface available in java.lang package.
2. All Wrapper classes and String class implements Comparable interface.

3. It is ment for default natural sorting order (DNSO).
4. It has the following abstract method:

public int compareTo(Object obj)

Current Object - Given object

→ +ve (Current Object > given Object)
→ -ve (Current object < given Object)
→ 0 (Current object = given object).

Constructors of Treeset:

1. TreeSet t = new TreeSet(); //DNSO.

2. TreeSet t = new TreeSet(Comparator c); //CSO

3. TreeSet t = new TreeSet(Collection c);

Example: (3)

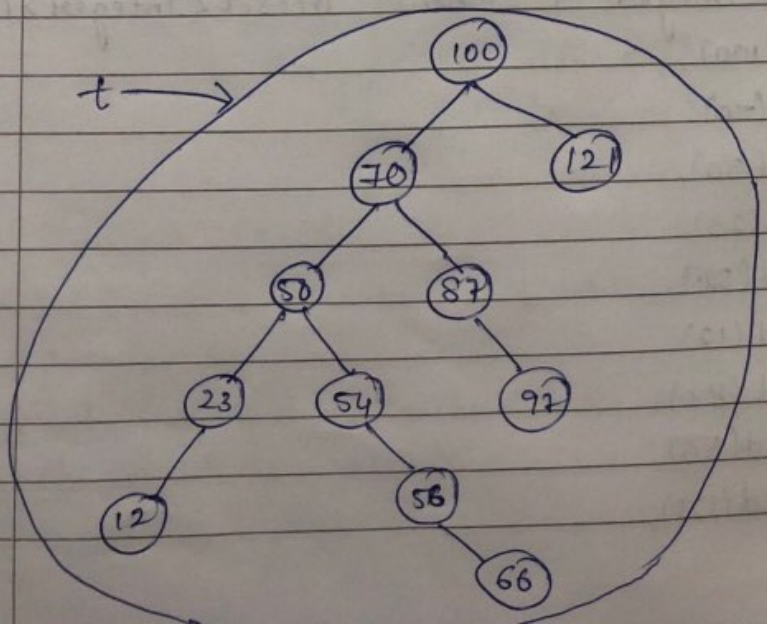
```
P.S.V.M(String[] args)
{
    TreeSet<Integer> t = new TreeSet<Integer>();
    t.add(100);
    t.add(70);
    t.add(50);
    t.add(23);
    t.add(54);
    t.add(12);
    t.add(87);
    t.add(56);
    t.add(121);
```

```
t.add(97);
t.add(66);
t.add(66);
//t.add(null); not allowed.

Sop(t);
Sop(t instanceof Cloneable); //true
Sop(t instance of Serializable); //true
Sop(t instance of RandomAccess); //false.
  }
}
```

t.add(100);
t.add(70); (70). CompareTo(100)
70-100

t.add(50); (50). CompareTo(100) 50-100 (4)
(50). CompareTo(70) 50-70

t.add(23); (23). CompareTo(100)
(23). CompareTo(70)
(23). CompareTo(50)



Inorder

Comparator (I):

1. It is available in java.util package.
2. It is ment for Customized Sorting Order.
3. It is not a functional interface.
4. It has following 2 abstract methods.

a) public int Compare (Object Obj1, Object Obj2);
   (Obj1: Current Object; Obj2: Given Object.)

b) public boolean equals (Object Obj);

Example:

```
package ban.cfl;
import java.util.Comparator;
import java.util.TreeSet;
public class TreeSetDemo2 implements Comparator<Integer>
{
 public static void main(String[] args)
 {
  Treeset<Integer> t = new TreeSet<Integer>(new TreeSetDemo2());
  t.add(100);
  t.add(150);
  t.add(80);
  t.add(70);
  t.add(110);
  Sop(t);
 }
 @Override
 public int Compare(Integer itr1, Integer itr2)
  {
  // return itr2 - itr1;
    return itr2.CompareTo(itr1);
  }
}
```

05/10/2018
Friday.

ArrayList Asc order:

```
Collections.Sort(al);
```

Arraylist Desc Order:

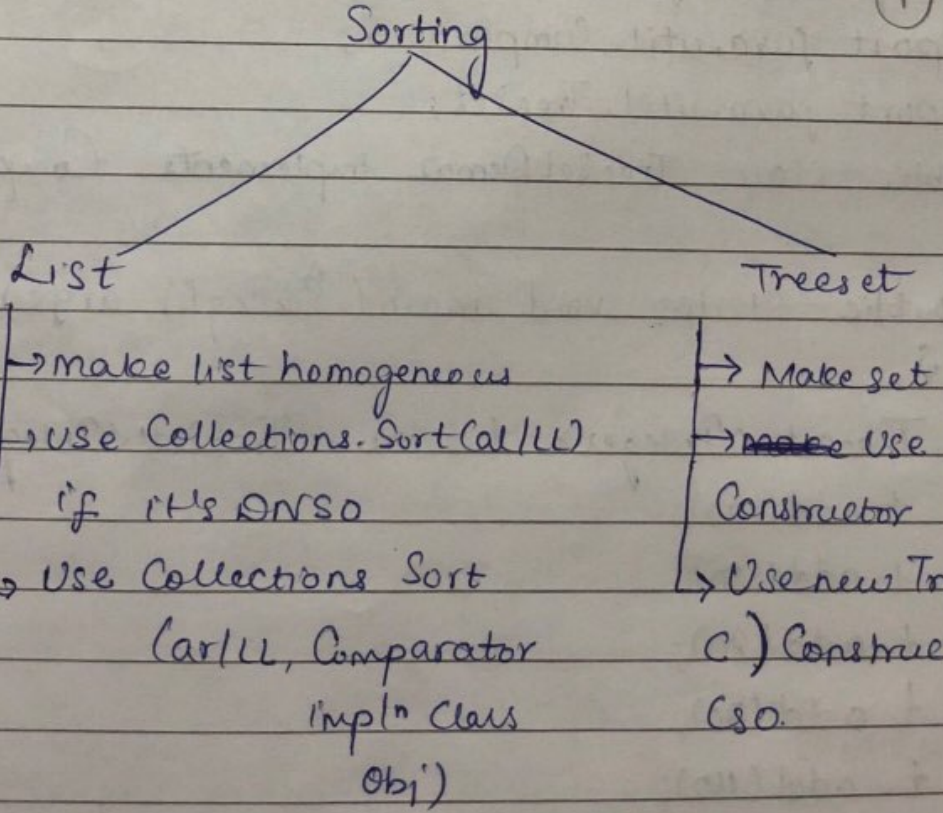
Implements Comparator

```
Collections.Sort(al, new <Clasname>());
```

Method (overridden)

```
public int compare(Integer itr1, Integer itr2)
{
  return itr2-itr1;
}
```

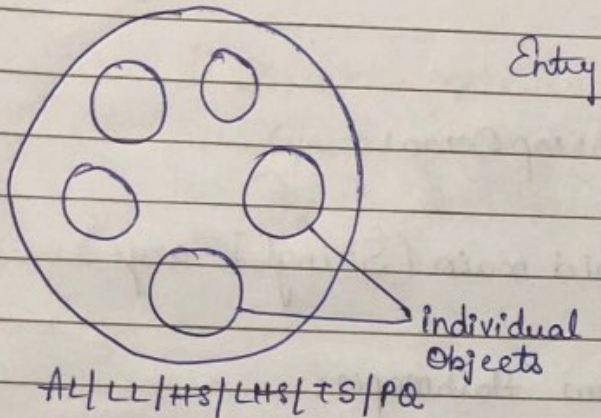
①



MAPS:

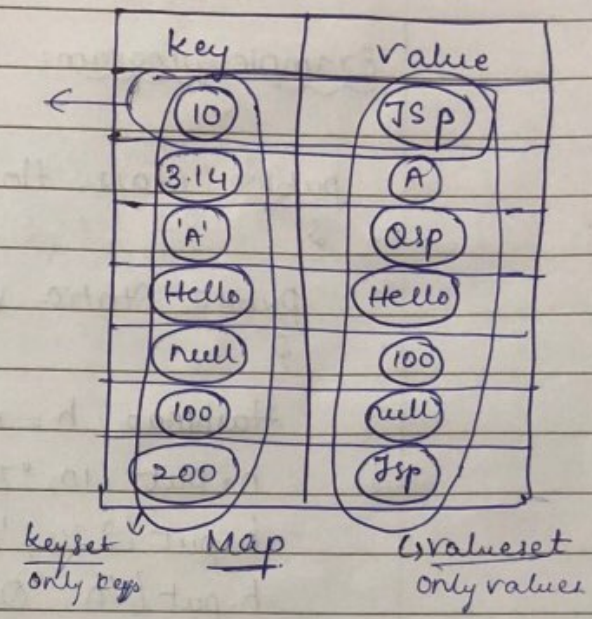
Definition:

If we want to represent a group of key-value pair of objects then we will go for Map.



~~AL/LL/HS/LHS/TS/PQ~~

Collection



(2)

1. Both key and values represents Objects.
2. Both key and values can be hetegrogeneous.
3. Both key and values can be same.
4. Null is posible for both key and value.
5. Duplicate values are allowed but not duplicate keys.
6. A pair of key-values is technically called as an 'Entry'.

Map interface Methods:

1. public void Object put(Object key, Object value)
2. public void putAll(Map m)
3. public boolean containskey(Object key)
4. public boolean containsValue(Object value)
5. public Object get(Object key)
6. public void remove(Object key)
7. public void clear();
8. public int size();
9. public boolean isEmpty();

10. public Set keySet()
11. public Collection values()
12. public Set entrySet()

Example program:

```
public class HashMapDemo1
{
 public static void main (String[] args)
 {
  HashMap h = new HashMap();
  h.put (10, "Jsp");
  h.put (3.14, 'A');
  h.put ('A', "Qsp");
  h.put ("Hello", "Hello");
  h.put (null, 100);
  h.put (100, null);
  h.put (200, "Jsp");
  Sop (h);
  h.put (100, "world");
  Sop (h);
  Sop (h.containskey (100)); //true
  Sop (h.ContainsValue ('A')); //true
  Sop (h.Set ('A')); //Qsp
  Set S1 = h.keySet();
  Sop ("Only keys :" + S1);
  Collection c = h.values();
  Sop ("Only values :" +c);
  Set S2 = h.entrySet();
  Sop ("Entries: "+S2);
 }
}
```

3

07/10/2018
SUNDAY.

interface map
{
(12) Methods

Interface Entry
{
i) Object getKey()
ii) Object getValue()
iii) Object setValue(Object obj)

} Only for Entry Objects.

}

Topic wise Java Questions.

Classes & Objects.

1. What is a class write the class template?
2. What is an Object.
3. What is byte code file? How many byte code files will be generated after Compiling source file.
4. What will be the name of the byte code file?
5. Command used to run & Compile java program
6. Can I declare local variable as static?
7. Does java allows global variable?
8. How many types of variables do java Supports?
9. How to access static & non static members explain with an example?
10. When to declare data member as static & Non-static.

Blocks.

1. What is block and how many types of blocks available?
2. Diff. b/w block & method
3. When static block will execute?
4. When non-static block will execute?
5. How many times static block will execute?
6. How many times non-static block will execute?
7. Can I initialize static data members inside non static block?
8. Can I initialize non-static member inside static block?

Constructors.

1. What is a Constructor & why is it used for?
2. When Constructors will be executed? Explain
3. When to go for argument Constructor? Explain with an example?
4. WAP to demonstrate allow return type? If we give what

4. WAP to demonstrate Constructor overloading & what are its advantages.
5. Does the Constructor allow return type? If we give what happens.
6. Can the Constructor be Static?
7. Can the Constructor be final?
8. What is the access modifiers allowed for Constructor?
9. What happens when we create an object explain Series of Steps.

Method Overloading:

1. What is method overloading explain with an ex?
2. Can we overload Static methods? WAP to overload main method.
3. Explain how to pass arguments to main method explain?
4. WAP to overload non-Static methods?
5. What are the advantages of method overloading.

Inheritance:

1. What is inheritance & what are its advantages?
2. Explain types of inheritance.
3. Explain why multiple inheritance is not allowed in java.
4. What is 'this' keyword? why is it used for? Explain with an example.
5. What is Super keyword? Why is it used for? Explain with an example.
6. Explain Constructor Calling with an example. what are its advantages.
7. Explain Constructor Chaining with an example? why Constructor Chaining.
8. Diff b/w this, Super, this(), Super().

9. What is the first statement inside default constructor?
10. What happens if a class is declared as final? Example for final.
11. Explain diamond ring problem.
12. Is object class a final? Justify.
13. Explain what happens when we instantiate Subclass.

Packages:

1. What is a Package? How to create it?
2. Where to write package creation statement.
3. Can one source file contain many package creation stmts?
4. When to use import? Explain with an example.
5. Can one source file contain many import statements.
6. In which order package and import keyword should exist in a source file?
7. Explain access modifiers provided by java?
8. What is data hiding? How to achieve it?
9. What is Encapsulation.
10. What is jar file? Where is rt.jar present? What it Contains?
11. Does default member gets inherited across the package?
12. Can I access public members outside the package if the class is default?
13. Can I declare a class a private? What access modifiers can I provide for a class.
14. What is the name of byte code file of inner class.

Abstract class and Abstract methods:

1. What is abstract method?
2. What is abstract class?
3. Where we provide definition for the abstract methods and what if we failed to provide?

4. When to declare a method abstract?
5. When to go for abstract class?
6. Can abstract class be final? Why?
7. Can I declare static method as abstract? Why?
8. Can I change the access modifieres while providing implementation.
9. Can the abstract class be intatiated? Why?
10. Does abstract class allows constructors. If yes, explain with an example, its usage.
11. Does abstract class allows static members. If Yes how to access them? Write example.
12. Can I declare private method as abstract? Why?
13. Advantages of Abstract class.

String Class and its methods.

1. What is String class? Where is it present?
2. What is the relationship between String & Object class?
3. What methods of object class overridden in String class & how?
4. Explain Constructors of String class?
5. How to convert array of Characters to String?
6. What is immutable? Why String is immutable?
7. Justify how String is immutable.
8. What are the advantages and disadvantages of immutable.
9. What is difference b/w String, StringBuilder & String Buffer?
10. Explain when the Objects will be created in SCP & heap memory?
11. Which methods of Object class overridden in StringBuffer & StringBuilder?
12. What is diff b/w final and immutable.
13. Why String is final?

14. Why do you think there are 2 ways of creating object of String class
15. How to create our own immutable class.

Object class and its methods

1. What is Object class? Where is it present?
2. Is Object class final? Justify?
3. Mention the methods object class.
4. Which methods of object class cannot be overridden?
5. Explain the behaviour of toString(), hashcode() & equals(Object obj).
6. WAP to Compare whether two mobiles are same or not w.r.t name, price & ram. Program should also display states of an object.
7. WAP to Compare whether two cars are same or not wrt name, price & color. Program should also display states of an object.
8. WAP to Compare whether two watches are same or not. Program should also display states of an object.
9. What is the difference b/w class & Class?
10. How to identify fully qualified name of the class when we have the object.

Exception Handling:

1. What is exception? Why to handle it?
2. If user don't handle exception, then who will handle?
3. What's benefit of user handling exception?
4. Why we shouldn't allow default exception handlers to handle exception.

5. Explain exception hierarchy in java.
6. How many handlers does java provides to handle exception.
7. Explain try catch block with an example.
8. What statements we should write in the try block?
9. Explain what happens when exception occurs in the try block.
10. Does catch block execute always?
11. Can one try block have multiple catch blocks? If yes which catch block will execute? What happens if the corresponding catch block not found?
12. Can we write generalized catch block & Specialized catch block together?
13. How many exceptions occur in single try block? How to handle more than one exception?
14. Explain multiple try catch blocks.
15. What is the use of throw keyword.
16. Explain the difference between JVM creating an exception object and user creating an exception object?
17. Explain exception propagation? When exception object will propagate?
18. What is Checked exception & Unchecked Exception?
19. When to use throws? Explain with an example.
20. Difference between Checked exception & Unchecked exception?
21. Difference between throw & throws.
22. Difference between throw & Throwable.
23. What is the importance of finally block? Explain with an example.
24. Difference between final, finally & finalize()
25. WAP to demonstrate custom Checked exception.
26. WAP to demonstrate custom unchecked exception.