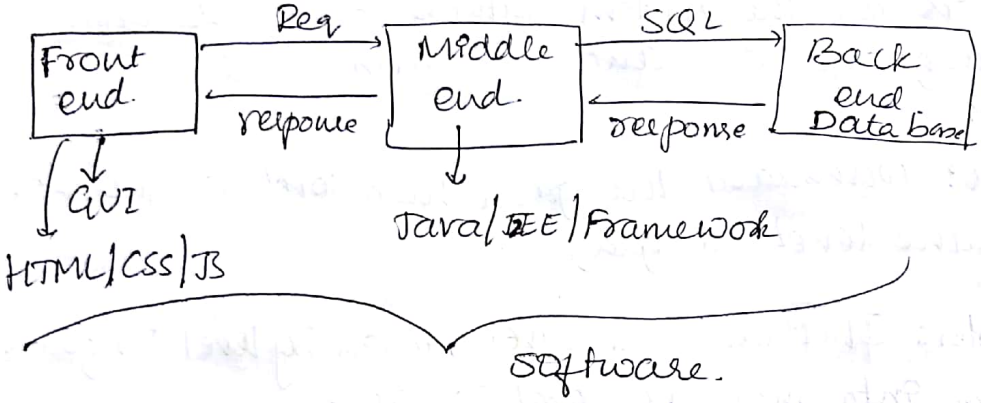
17/08/17. Java Thursday.



Software.

→ S/w Required.

① JDK (Java development kit)

* 1995 — 1.0V 1.8V
* Sun micro system owned initially, introduced by James Gosling who is the worker of SMS.
* Takeover by ORACLE.
* SCJP / OCJP. → Exam.

② Editor

* Note pad * wordpad * Textpad * Notepad++
* Editplus ✓

③ IDE

* Eclipse ✓ → Neon, Oxygen, Mars
* Net bean

→ Java → Stand editor (practice)
→ Enterprenem editor (Enterprise)
→ Micro editor (Cellphone)

→ program file → JDK → bin (Copy)
→ My comp → advanced setting → env. var → Sys variable (Rightclick)
→ path → after path ; paste ;.

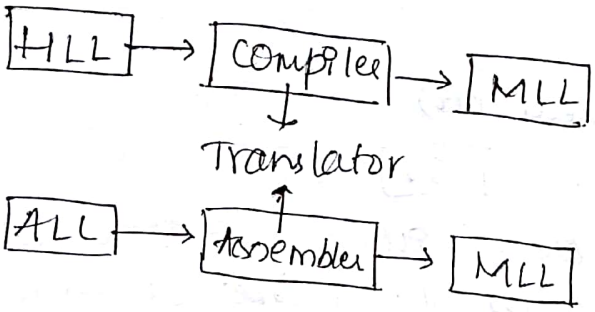
* TRANSLATORS:

Translators are the system software which translates one form of language to another form.

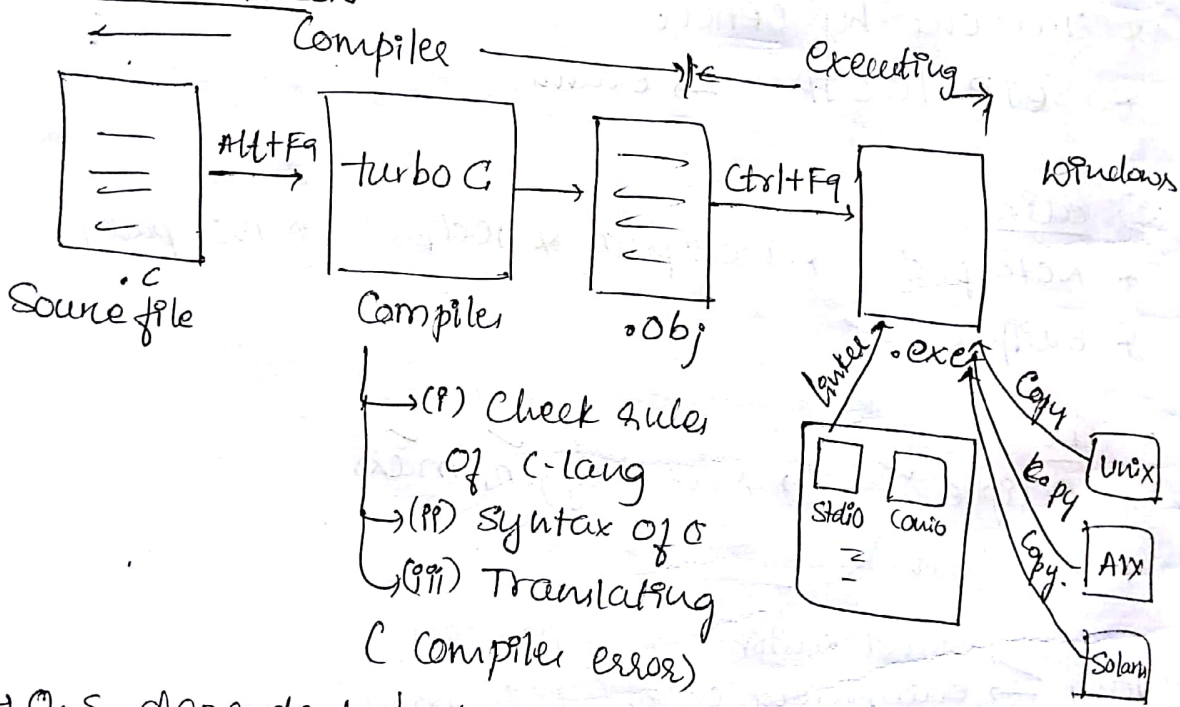
2 types:

① Compiler: Translates the given high level language program into machine level language

② Assembler: It translates given Assemby level language program into machine level language



* C- COMPILATION

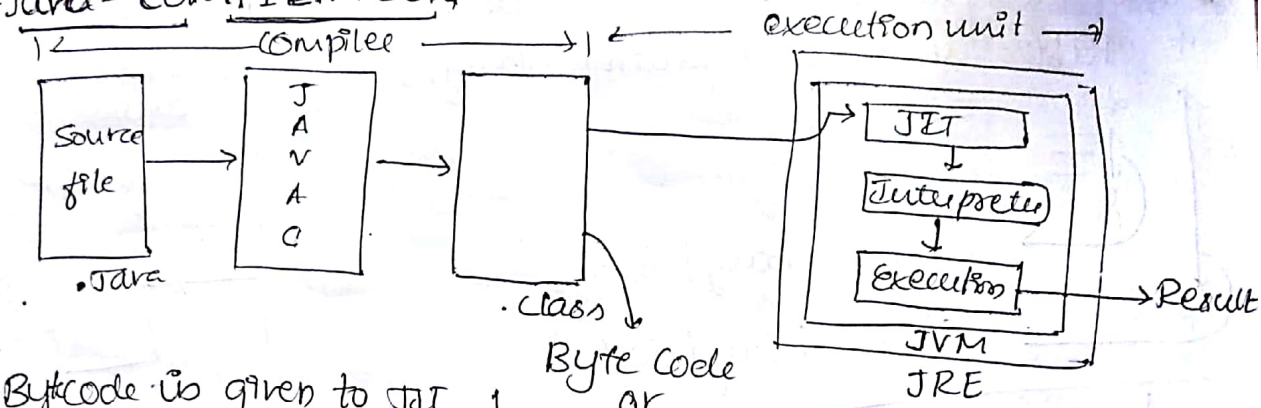


→ O.S dependent / platform dependent, program written on one O.S can't be run in other O.S. So we should write the program on other O.S again and execute it

→ When there is mistake in built in functions (printf, scanf etc) linker will give error. But Compiler compiles successfully since it only check for rules.

→ faster since directly interacts with Hardware through pointer which is not supported by Java.

*Java - COMPILATION



Bytecode is given to JIT Interpreter Compiles taking one by one instruction and gives to execution block which give Result.

Byte Code
or
Execution file
or
Class file
or
Intermediate file

JVM → Java virtual m/c.

JRE → Java Runtim environment

JIT → Just In Time Compiler.

↔ What is JVM?

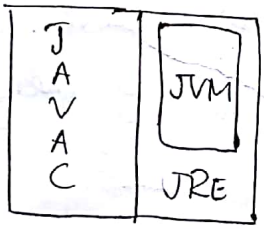
It is a software which perform the operation like a physical machine, but its physical existance is not there.

→ What is JRE?

It is a software which provides the necessary environ-ments required to execute the java program.

→ What is JDK?

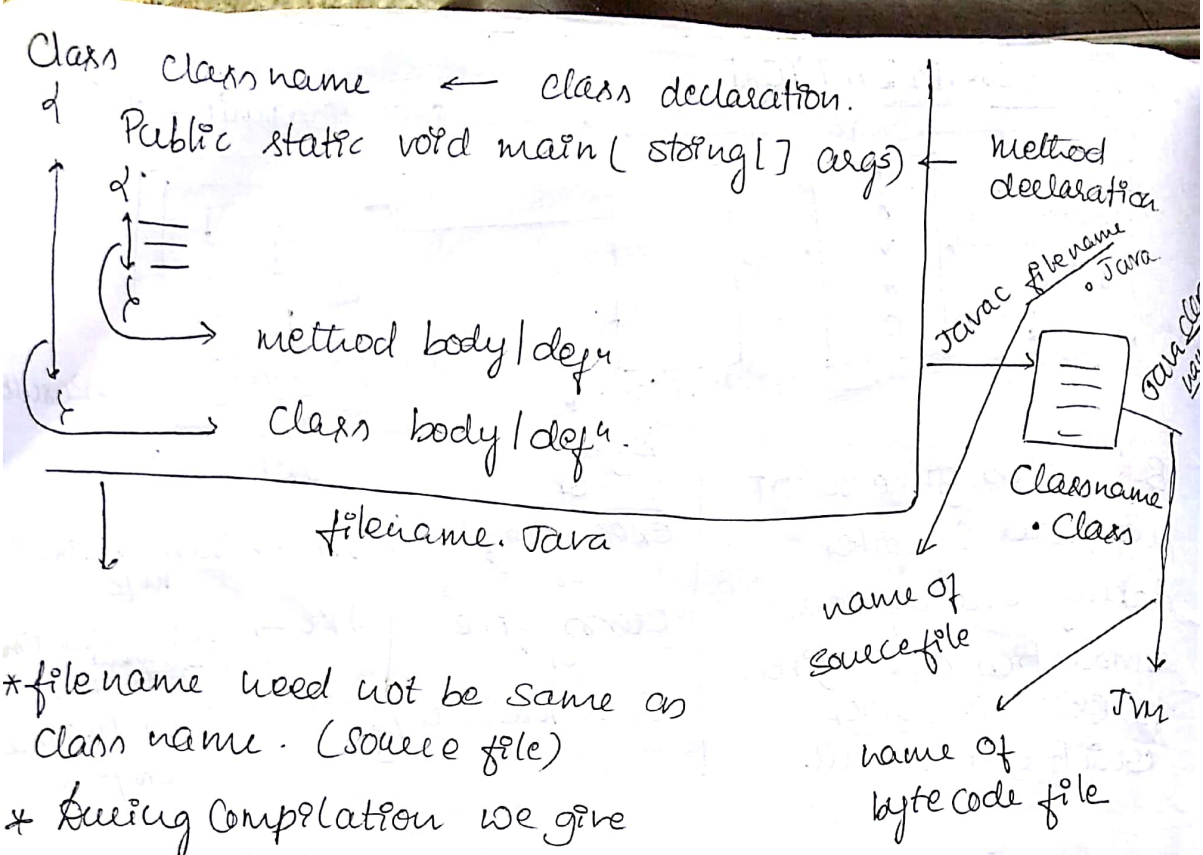
It is a software which contains all the necessary components required to compile and execute the Java program.



JDK
(Java Development Kit)

21/08/17

* Structure of Java Program.



* filename need not be same as class name. (source file)
* During compilation we give source file (Javac filename.Java)
* During execution we give bytecode file (Java classname.~~Java~~)
* Byte code file name should be same as class name.

Ex: ① Class test
{
Public static void main (String[] args)
{
System.out.println ("Hello world~~!~~");
}
}

→ Save in a particular drive, file → New → .Java.
→ Change directory CD: and keep on giving link.
→ Change drive D:

② Class program
{
Public static void main (String[] args)
{
System.out.println ("Hello world");
System.out.println ("welcome");
System.out.print ("Hello world");
System.out.print ("welcome");
}
}

O/P: Hello world
welcome
Helloworld Welcome.

Bcz \n takes to next line

③ Class program2
{
Public static void main(String[] args)
{
System.out.println("Hello\n world\n Hello\n Guys");
System.out.println("Hello\t world\t Hello\t Guys");
}
}

O/P: Hello
World
Hello
Guys
Hello World Hello Guys

* '+' operator in Java:
This is the only operator in Java which possess dual functionality
→ Adds 2 Characters and Numeric values
→ Concatenate the strings without space.

Ex: ① 10 ⊕ 20 → add ⇒ 30
② "Hello" ⊕ "World" → concat ⇒ Helloworld.
③ "Hello" ⊕ 10 ⇒ Hello 10
④ 10 + "Hello" ⇒ 10Hello.
⑤ "Hello" 10 + 20 ⇒ Hello 1020
⑥ 10 + 20 + "Hello" ⇒ 30Hello.
⑦ "A" + 'B' ⇒ AB.
⑧ 'A' + 'B' ⇒ 131
65 66.

Keywords

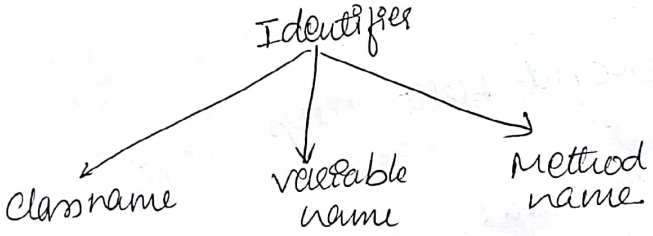
There are the set of reversed words defined by the programming language.

In Java all keywords are in Lower case

Eg: static, public, void, char, interface, default, final abstract, super etc

Identifiers: *Used by the programmer to identify the elements of programe.

* These Identifiers can be a class name or a variable name or a method.



* Rules for creating identifiers:

1) An identifier can have alphanumeric characters.
2) An identifier can consist of both uppercase and lowercase characters.
3) An identifier should not start with a digit.
4) Spaces in between the identifier is not allowed.
5) The only special symbol allowed in identifier creation is underscore (_) & $ is also allowed in Java but very seasly used.

Ex: (1) Class Test01 ✓
(2) int Sumof2numm ✓
(3) Class 1demo ✗
(4) int mul 2 num ✗
(5) addTwonumc) ✓
(6) add_three_number ✓

→ It standards of creating an identifier

(1) First word first character is in upper case; successive first character should also be in upper case.
Class Class
Ex: SavingsAccount, CurrentAccount, ~~EmployeeName~~ etc

(2) For variables and methods.
All the characters of first word should be in lowercase, First Characters of successive words should be in uppercase.

Ex: addTwoNumbers(), SubTwoNum(), withDraw()
empId, amtBal etc.

* Variable:

It is an identifier used to store some data or
It is an identifier used to represent memory location.

Eg: int a;
double d;
Char c;
String s;

Declaration.

Data_type identifierName;

- Data_type → primitive / Non primitive. (Class type or Interface type)
- identifierName → variable Name.

Assignment / Initialization

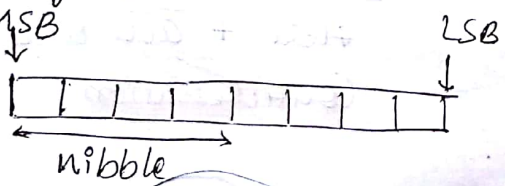
variable name = value;

i = 100, d = 8.14, c = 'A', s = 'Spiders';

* Primitive Datatypes in Java. (8)

① Numeric
- Integer
  - byte(1)
  - Short(2)
  - int(4)
  - long(8)
- Floating
  - float(4)
  - double(8)

② Non Numeric
- Char (2) (In Java)
- boolean (1 bit)



1 byte = 2 nibble = 8 bit.

Ex: int i;
double d;
char c;
float f;
long l;
boolean b;
string s;
} declaration

i=10
d=8.14;
c='z';
f=8.18f;
l=99999l;
} Initialization or assignment

b= 1 ; error
b = 0 ; error
b = 'true'; error
b = 'false'; error

b= flase;
b = true;
s = 'true';
d = 'false';
s= true;
s= false;

f and l are used in float & long to differentiate bt float, double & int & long.

23/08/17

Ex: Program:-

```
class program1
{
  public static void main (String[] args)
  {
    System.out.println ("main method started")
    // declaration
      int i;
      double d;
      float f;
      boolean b;
      char c;
      String s;
    // Initialization / Assignment
      i= 100;
      d= 434.33;
      f = 465.23f;
      b = true;
      s = "Jspiders";
      c = 'z';
      S.O.P ("i = "+i);
      S.O.P ("d = "+d);
      S.O.P ("f = "+f);
      S.O.P ("b = "+b);
      S.O.P ("s = "+s);
      S.O.P ("c = "+c);
  }
  S.O.P ("main method ended");
}
```

O/P:

i= 100
d= 434.33

Here + act as Concatenation.

*A variable can be re-initialized any number of times but can't be redeclared within the same method.

Ex program 2

```
class prgm2.
{
  ~~class~~ public static void main(String[], args)
  {
    S.O.P ("main method started");
    int i=10;    // declaration.
    S.O.P ("i= "+i);
    i=20         // redeclaration (Re-initialization.
    S.O.P ("i= "+i);
    i=30
    S.O.P ("i= "+i);
    // int i = 40; not allowed bcz declaration is (error)
                   not allowed.
    S.O.P (" main method ended");
  }
}.
```

* If a variable is declared final, such variables should be initialized only once. i.e., for a final variable, Re-initialization is not possible.

Ex prgm 3

```
Class prgm3
{
  public static void main (String[] args)
  {
    S.O.P (" main program started");
    final int i=10 ; // Initialization.
    i=20 ; // error, bcz i is final which cannot
              be re-assigned.
    S.O.P (" Main method ended");
  }
}
```

* In Java any variable can't be used without initializing. that gives error.

Ex program 4

```
class program4
{
    public static main(String[] args)
    {
        S.O.P("main method started");
        int i;
        S.O.P("i= "+i);
        S.O.P("main method ended");
    }
}
```

// Error.

```
int i;
j=i+10;
SOP(j); //Error
```

→ Operators

* Arithematic * Relational * Logical * Shift
* Bitwise * Assignment * Ternary * Unary

* Used to perform specific operations.
* Used to Build the expressions.

ex: SOP(10/5); //2
SOP(10%5); //0

* Arithematic Precedence

① int x;
x = 10+20;
LHS RHS
x=30;

② int x;
x = 10+20*3+2;
= 10+60+2
x = 72;

③ int x;
x = 10*3-2/2+10*5/2
= 30-2/2+10*5/2
= 30-1+10*5/2
= 30-1+50/2
= 30-1+25
= 29+25
x = 54

+ -
/ * %

④ int x;
x = 10+2*3
= 10+6
x=16;

⑤ int x;
x = (10+2)*3
= 12*3
x = 36;

* Relational (<, >, <=, >=, !=, ==)

Result of relational operator will always be a boolean value.

Ex: ① int x=10, y=20;

x>y ⟹ false
x<y ⟹ true
x!=y ⟹ true
x==y ⟹ false

② int x=10, y=10

x==y ⟹ true
x>=y ⟹ true
x!=y ⟹ false
x<y ⟹ false.

* Logical (&&, ||)

Logical operator depends upon the result of relational operator.

Ex: ① int x=10, y=20;

x>y || x==y
false || false ⟹ false

x!=y && x<y
True && True ⟹ true

* Bitwise (&, |)

Ex: ① int x=10, y=10;
int z = x&y; // 10
int m = x|y; // 10
int n = x&5; // 0

1010
(&) 1010
1010 → z

1010
(|) 1010
1010 → m.

1010
0101
0000 → n.

28/08/17

* Unary operator: It is an operator which expects a single operator. It can be classified as.

(1) Increment (++) → Post, pre
(2) Decrement (--) → post, pre.

Ex:

① int x = 10;
int y = x++;
S.O.P(x); // 11
S.O.P(y); // 10

x: 10 11
y: 10

int x=10;
int y=++x
S.O.P(x); //11
S.O.P(y); //11

x: 10 11
y: 11

② int x = 10;
int y = x++ + x++;
10 + 11
S.O.P(x); //12
S.O.P(y); //21

x: 10 11 12
y: 21

int x=10;
int y = ++x + ++x
11 + 12
S.O.P(x); //11
S.O.P(y); //23

③ int x=10;
int y = x++ + ++x;
10 + 12
S.O.P(x); //12
S.O.P(y); //22

x: 10 12
y: 22

④ int x = 10;
int y = x++ + x++ + ++x;
10 + 11 + 13
S.O.P(x); // 13
S.O.P(y); // 34

x: 10 13
y: 34

⑤ int x=10;
int y = ++x + x-- + x++ + --x + x--;
S.O.P(x); // 11 + 11 + 10 + 10 + 10
S.O.P(y);

x: 9
y: 52

② class progm:

```
{
public static void main(String[] args)
{ S.O.P(" main method started");
  int x = 10;
  S.O.P(x++);          // 10
  S.O.P(x++ + ++x);    // 25 24
  S.O.P(x++ + x++ + x--);   // 42
  S.O.P(++x + x++ + --x);   // 45
  S.O.P(x++ + ++x + x++);   // 49
  S.O.P(x++);          // 18
  S.O.P("main method ended");
}
}
```

* CONTROL STATEMENTS :-

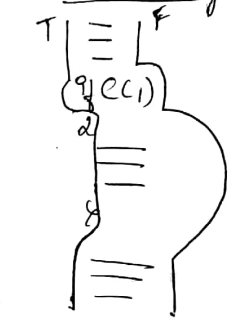
→ These are used to control the flow of execution of the program.

→ Control statements are classified as follows.

① Branching
- If
- If else
- nested If
- If else ladder
- switch

② Looping.
- For
- While
- Do-While

① Branching :-



if-else ladder

```
if (c1)
{
}
else if(c2)
{
}
else if(c3)
{
}
else
```

nested-if

```
if (c1)
{
  if(c2)
  {
  }
  else
  {
  }
}
else
{
  if (c3)
  {
  }
```

switch(value)

```
{
case 1: ==
    break;
case 2: ==
    break;
default: ==
}
```

switch

→ In Switch statement, it is not mandatory that default should be in bottom.

29/08/17

Eg:1

```
class program1
{
  p s v m (String [] args)
  {
    S.O.p ("main method started");
    int num = 4;  // initialization
    if (num>8)
    {
      S.O.p (num+" is greater than 8");
    }
    else if (num<8)
    {
      S.O.p (num+" is lesser than 8");
    }
    else
    {
      S.O.p (num+" is equal to 8");
    }
    S.O.p ("m m ended");
  }
}
```

Eg 2:

```
class program2
{
  P s v m (String[] args)
  {
    S.O.p(" main method started");
    double amtBal = 50000.00;
    double withDraw = 60000.00;
    if (amtBal>withDraw)
    {
      S.O.p (" Successful withdraw");
      amtBal = amtBal - withDraw;
    }
    else
    {
      S.O.P (" Insufficient Balance");
    }
    S.O.P (" Balance = " + amtBal);
```

Program 3:

```
class prgm3
{
  PSVM (S[] args)
  {
    S.O.P ("Main Method started");
    double amtBal = 50000.00;
    double WithDraw = 20000.00;
    int pin = 1234;
    if (pin == 1234)
    {
      SOP ("valid pin");
      if (amtBal > WithDraw)
      {
        S.O.P ("Successful withdraw");
        amtBal = amtBal - withdraw;
      }
      else
      {
        S.O.P ("Insufficient Balance");
      }
        S.O.P ("Balance = " + amtBal);
    }
    else
    {
      S.O.P("Invalid pin");
    }
      S.O.P ("Main method ended");
  }
}
```

Progrm 4

```
class progrm4
{
  PSVM(S[] a)
  {
    SOP ("Main method started");
    char grade = 'A';
    switch (grade)
    {
```

```
        case 'A': S.O.P ("FCD");
                  break;
        case 'B': S.O.P ("FC");
                  break;
        case 'C': S.O.P ("SC");
                  break;
        case 'D': S.O.P ("Fail");
                  break;
        default : SOP ("Invalid grade");
    }
    SOP ("Main Method ended");
  }
}
```

if-else Cases

① `int x=10;`
```
if (x == 10)
{
  ===
}
else
{
  ===
}
```

② `int x= 10;` — error
```
if (x = 1)
{
  ===
}
else
{
  ===
}
```

③ `if (1)` — error
```
{
  ===
}
else
{
  ===
}
```

④ `if (true)`
```
{
  ===  ✓
}
else
{
  ===
}
```

⑤ `if (false)`
```
{
  ===
}
else
{
  ===  ✓
}
```

⑥ `if ("true")`
```
{
  ===
}
else → error
{
  ===
}
```

⑦ `if ("false")`
```
{
  ===
}
else
{
  ===
}
```

⑧ `String s = "true"`
```
if (s)
{
  ===
}
else
{
  ===
```

error

⑨ Boolean b= true;
if (b)
{
==== ✓
}
else
{
====
}

⑩ if ()
{
====
}
else
{
====
} Error

⑪ if (true)
SOP ("Hello");
SOP ("World");
else
if (true)
{
SOP ("Hello");
}
{
SOP ("World");

Olp: Hello World.

⑫ if (false)
SOP ("Hello");
SOP ("World");

Olp: World.

* LOOPING x STATEMENTS

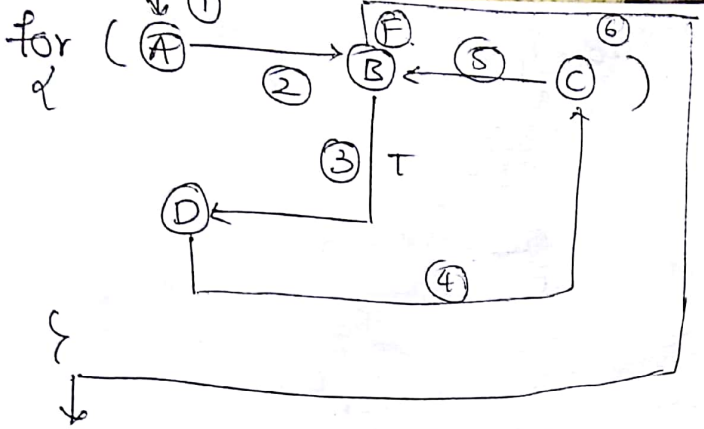
1. For loop

```
int i=1;
SOP(i); //1
i++;
SOP(i); //2
i++;
SOP(i); //3
i++;
SOP(i); //4
```

```
for (int i=1; i<=5; i++)
{
  SOP(i);
}
```

SYNTAX:

```
FOR(initialization; condition; inc/dec)
{
  ====   // Body of loop.
}
```



Ex ① for (int i = 5; i >= 1; i--)

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

O/P: 5 4 3 2 1

② for (int i=15; i<=25; i++)

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

O/P: 15 16 ⋮ 25

③ for (int i=25; i>=15; i--)

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

O/P: 25 24 ⋮ 15

30/08/17

① for (int i=1; i<=5; i++) # of rows.

```
{
  for (int j=1; j<=5; j++)   # of columns.
  {
    S.O.print(j);
  }
  S.O.println();
}
```

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

Instead of

```
for (int i=1; i<=5; i++)
{
  S.O.print(i);
}
  S.O.println();
```

5 times

②
```
for (int i=1; i<=5; i++)
{
    for(int j=1; j<=5; j++)
    {
        S.O.print (i);
    }
    S.O.println( );
}
```

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

③
```
for (int i=1 ; i<=5 ; i++)
{
    for (int j= 5; j>=1; j--)
    {
        S.O.print (j);
    }
    S.O.println ();
}
```

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

④
```
for (int i=5; i>=1; i--)
{
    for (int j=5; j>=1; j--)
    {
        S.O. print (i);
    }
    S.O. println ();
}
```

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

⑤
```
for (int i= 1; i<=5; i++)
{
    for (int j=1; j<=i; j++)
    {
        S.O. print (j);
    }
    S.O.println();
}
```

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

(12)

Olp:

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

* Different ways of writing For loop.

①
```
for (int i=1; i<=5; i++)
{
    S.O.P(i);
}
```

②
```
int i=1;
for( ; i<=5; i++)
{
    SOP(i);
}
```

③
```
int i=1;
for( ; i<=5; )
{
    SOP(i);
    i++;
}
```

④
```
int i=1;
for( ; ; )
{
    S.O.P(i);
    i++;
}
```

Olp: Infinite loop.

⑤
```
int i=1;
for (;;)
{
  S.O.P(i);
  i++;
  if (i>5)
  {
    break;
  }
}
```

⑥
```
int i=1;
for(;;)
{
  S.O.P(i) ##   1 2 3 4 5
  i++;
  if (i<=5)
  {
    S.O.P(i);
    break;
  }
  else
  {
    continue;
  }
}
```
should be used only in loop.

O/P: error

⑦
```
int i
for (i=1; i<=5; i++);
{
  S.O.P (i);
}
```
until 5<=5 it will keep executing b/w i<=5 & i++ When, 6<=5 for breaks and execution continues.

31/08/17

* while and do-while loop.

```
int i=1;
while (i<=5)
{
  SOP(i);
  i++
}
```

Syntax
```
while (condition)
{
  ==   Body of while loop.
}
```

```
int i=1;
do
{
  SOP(i);
  i++;
} while (i<=5);
```

Syntax:-
```
do
{
  ==   → Body
} while (condition);
```

Ex 1: int i=1;
while(i++<=5)
{
SOP(i);
}

1<=5 2<=5 3<=5 4<=5
o/p: 2 o/p: 3 o/p:4 o/p: 5
5<=5.
o/p: 6

Ex 2: int i=1;
while (++i<=5)
{
SOP (i);
}

2<=5 3<=5 4<=5 52=5
o/p: 2 o/p: 3 o/p:4 o/p:5

Ex 3: int i=1;
while (i++<=5);
{
SOP(i); //7
}

1<=5
2<=5
3<=5
4<=5
5<=5
6<=5

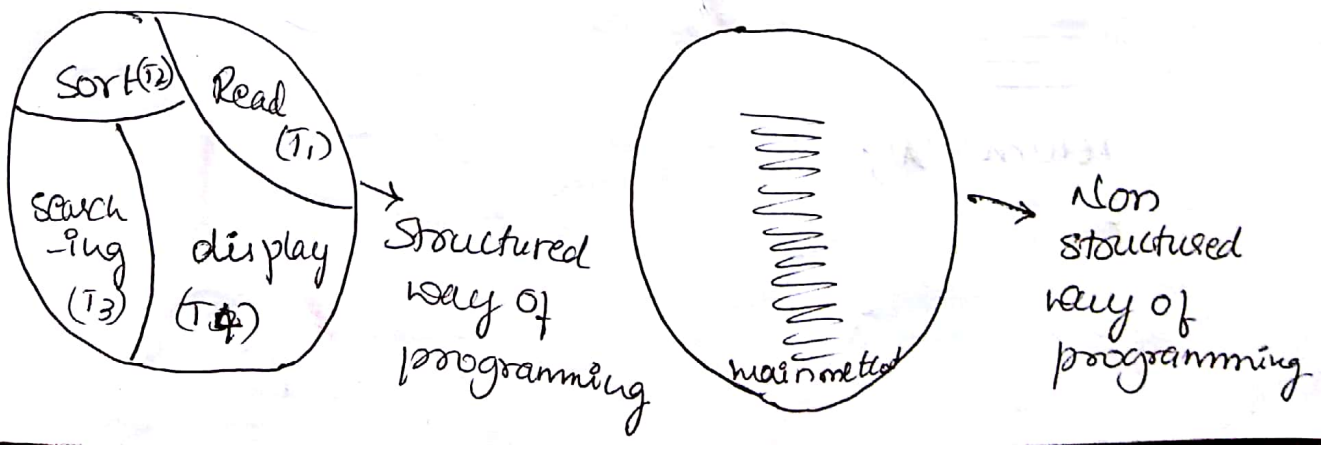
Ex 4: int i=1;
while (++i<=5);
{
SOP(i); //6.
}

* METHODS (FUNCTIONS)

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

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

→ ~~Increases~~ +



SYNTAX

<Access modifier> <modifier> Return type Method name (<Ar

{

≡≡≡ ↕ method body (def)

}

method declaration.

<> → Optional.

Ex: Public static boolean m1()

{

≡≡≡

return true;

}

Public static void m2()

{

≡≡≡

return;

}

Public static void m3()

{

≡≡≡

}

Public static int m4()

{

≡≡≡

return 100; (100 Values)

}

Public static Char m5()

{

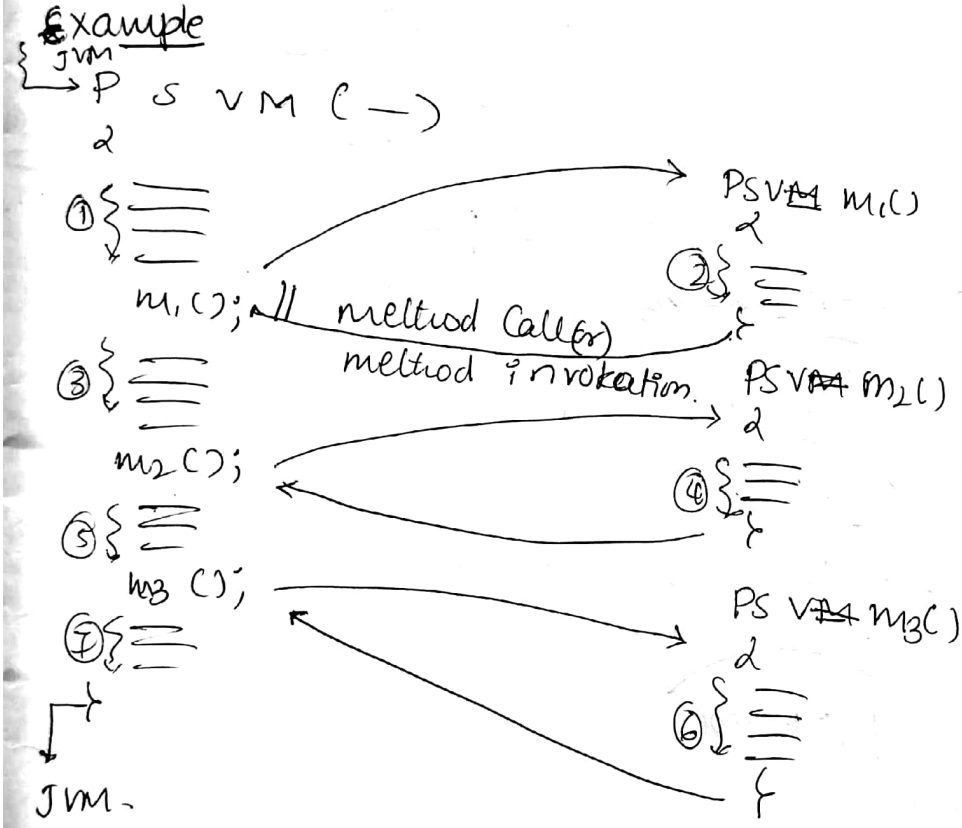
≡≡≡

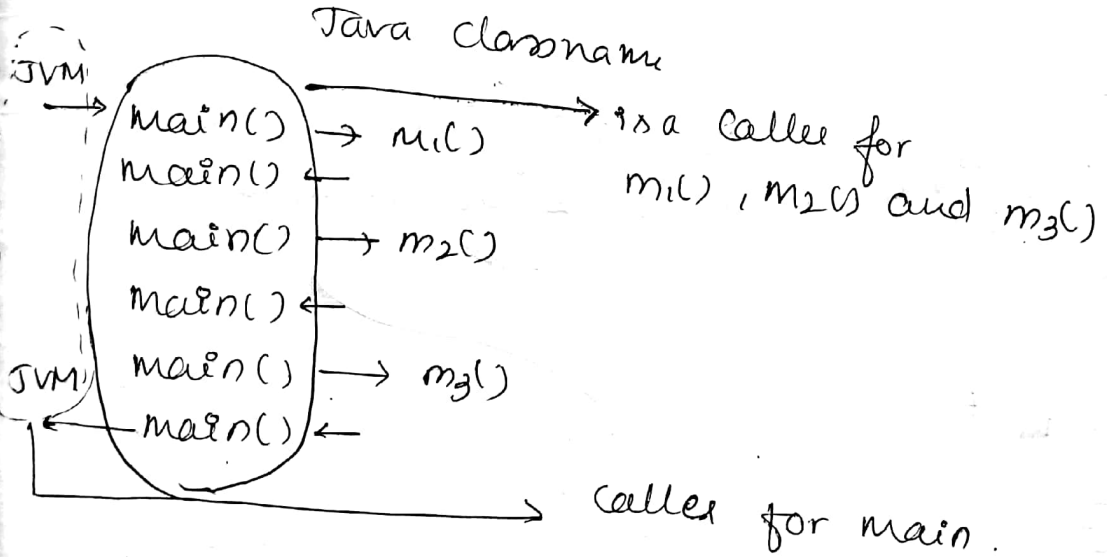
return 'A';

}

{
}
return 8.14f;
return 'A'; → this code can't be reached. Hence will get compile time error(code can't be reached).
}

Example





Example

JVM

↳ PSVM (—)

{

① ≡

m1(); // method call

⑦ ≡

}

JVM

PSV m1()

{

② ≡

m2(); ⑥

}

PSV m2()

{

③ ≡

m3(); ⑤

}

PSV m3()

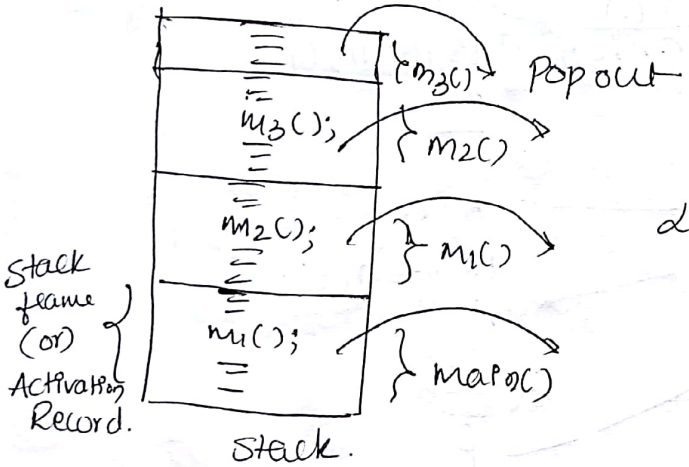
{

④ ≡

}

JVM → main() → m1() → m2() → m3()

JVM ← main() ← m1() ← m2() ← m3()



LIFO

01/09/17

* 4 types of Caller and Calling with arg & with return type
without arg & return type

① m1(); // method call

PSVM1()

{

≡

}

② Double x = m1(1000);

SOP(x);

PS double m1(int arg)

{

SOP(arg); // 1000

≡

return 80.00;

}

③ Boolean by = m1() without arg with return ④

sop (by); // true

```
PS Boolean m1()
{
    ==========
    return true;
}
```

m1('A'); with arg without ret

```
PS V m1(char arg)
{
    SOP(arg); //A
    ==========
}
```

Ex: ① class pgm1

```
{
  PSV M(String[] args)
  {
    S.O.P("Main Method started");
    S.O.P("Main method ended");
  }
  PSV S1()
  {
    S.O.P("running S1() method");
  }
  PSV S2()
  {
    S.O.P("running S2() method");
  }
  PSV S3()
  {
    S.O.P("running S3() method");
  }
}
```

O|P: Main Method Started
Main Method ended.

② class pgm2

```
{
  PSVM (String() args)
  {
    S.O.P("Main method started");
    S.O.P("Main method ended");
  }
}
```

m1();
m2();
m3();

O|P: Errors (3).

Ex3: Class program3

```
{
  P S V main(String[] args)
  {
    S.O.P(" Main method stasted");
    S1(); // method call
    S.O.P(" Main method ended");
  }

  PSV S1()
  {
    S.O.P(" S1() method stasted");
    S2();
    S.O.P(" S1() method ended");
  }
  PSV S2()
  {
    S.O.P(" S2() method stasted");
    S3();
    S.O.P(" S2() method ended");
  }
  PSV S3()
  {
    S.O.P(" S3() method stasted");
    S.O.P(" S3() method ended");
  }
}
```

O|P: Main method stasted.
S1 method stasted
S2 — " —
S3 — " —
S3 - " - ended
S2 - " ———
S1 —— " ——
main method ended.

Ex 4: write a program to add 2 numbers using method

```
Class program4
{
  PSVM (String[], args)
  {
    S.O.P ("Main method started");
    int num1=1, num2 = 423;
    int result = addtwonum (num1, num2);
    S.O.P ("Sum of "+num1+" & "+num2+" = "+result);
    S.O.P("Main method ended");
  }
  P.S int addtwonum ( int arg1, int arg2)
  {
    int temp; // declaration.
    temp = arg1 + arg2;
    return temp;
  }
}
```

O/P: M M S

~~10~~ Sum of 1 & 423 = 424

M M E.

Ex5: write a program to check whether number is odd or even. using method.

```
Class program5
{
  PSVM (String[], args)
  {
    S.O.P ("Main Method started");
    int num1 = 10;
    int result = addoreven (num1);
    ~~S.O.P ("num1" is even~~
    S.O.P ("Main method ended");
  }
  P.S int addoreven ( int arg)
  {
    if (num % 2 == 0)
    {
      S.O.P ( num+ " is even");
    } else
```

```
    SOP (num + " is odd");
  }
}
```

O/P: 10 is even;

* How to read i/p from user.

(1) import java.util.Scanner;

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

(3) For integer. → sc.nextInt();
For double → sc.nextDouble();
For long → sc.nextLong();
For char → sc.nextChar(); ×
sc.next().charAt(0); ✓
For String → sc.next(); // read until space
sc.nextLine(); // read along with space

In C

```
Scanf("%s", name); // to read until space
gets(name);        // read along with space
```

Ex:

```
class prgm1
{
  P S V M (String[] args)
  {
    S.O.P ("Main method started");
    // declaration
    String Coll_Name;
    String name;
    int id;
    double Marks;
    Char grade;

    Scanner sc = new Scanner(System.in);
    S.O.P ("Enter the College name");
    CollName = sc.nextLine();
    S.O.P ("Enter the name");
    name = sc.nextLine();
```

```
SOP("Enter the id");
id = sc.nextInt();
SOP("Enter the Marks");
marks = sc.nextDouble();
SOP("Enter The grade");
grade = sc.next().charAt(0);
SOP("Student details are !!!");
SOP("Name=" + name);
SOP("Id=" + Id);
SOP("Marks=" + marks);
SOP("grade=" + grade);
SOP("College Name=" + CollName);
S.OP("Main Method ended");
}
}
```

Assignment-2

→ Write a program to read 3 integer numbers from the user and perform addition using method.

→ Read a character from user and check whether it is vowel or not. using method.

04/09/17

→ Arrays are used to store homogeneous type of data
→ Array allocates consecutive memory location.

→ Declaration:-

```
Datatype[] arrayName;
```

primitive non-primitive (class type).

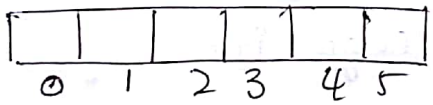
Eg: int[] intArr;
double[] doubleArr;
char[] charArr;
} primitive.
String[] args → non primitive.

→ Specifying size:-

```
arrayName = new Datatype[size];
```

Eg: intArr = new int[3];
doubleArr = new double[3];

Here data type should be same.



Subscript starts from 0, because Java strictly follows Numbering system rules. (Octal, deci, Hex start from 0).

→ Initialization.

```
arrayName[index] = value
```

Eg: intArr[0] = 10;
intArr[1] = 20;
intArr[2] = 30;
intArr[3] = 40; Exp: Array index out of boundary Exp.

Java strictly check types. But C allocates memory but that will be garbage.

Default arrayvalue
int = 0; double = 0.0;

→ Length: variable or property which is used to get the length of an array during run time dynamically

Syntax:

```
Arrayname.length
```

Ex program

```
Class pogm1
{
  Pu S V m(String[] args)
  {
    SOP ("main method started");
    int[] intArr; // Declaration
    intArr = new int[100] // specifying size.
    S.O.P ("Default values");
    for (i=0; i< intArr.length; i++)
    {
       S.O.P (intArr[i]);
    }
   // initializing Array
   for (int i=0; i< intArr.length; i++)
   {
     intArr[i] = i+100;
   }
   S.O.P ("After initializing");
   for (int i=0; i< intArr.length; i++)
   {
     S.O.P (intArr[i]);
   }
     S.O.P ("main method ended");
  }
}
```

To print in reverse

```
S.O.P("In reverse or
for(int i= intArr.length-
        i>=0; i--)
  {
    S.O.P (intArr[i])
  }
```

O/P:

Default values	After initializing
0	100
0	101
0	102
0	103
0	104

→ length() is used to get the length of the string.

Eg:

* int[] intArr;
* intArr = new int[100];
* String S = 'jspiders';
* intArr.length; ~ //100
* intArr.length(); error
* S.length(); ✓ //8
* S.length; error.
* int[] intArr = new int[5] // single line declaration.

* Passing Array to a method and Returning Array from the method.

To pass value to method

```
m1(8.14)

PSV m1(double arg)
{
  ≡≡
}
```

To pass value Array to method

```
int[] intArr = new int[5];
m2(intArr);

PSV m2(int[] intArr)
```

To return value from meth

```
String s = m2()
SOP(s); //jsp

PS String m2()
{
  ≡≡
  return 'jsp';
}
```

return array from meth

```
String[] s = m2();

PS String[] m2()
{
  String[] arr =
      new String[10
  ≡≡
  return arr;
}
```

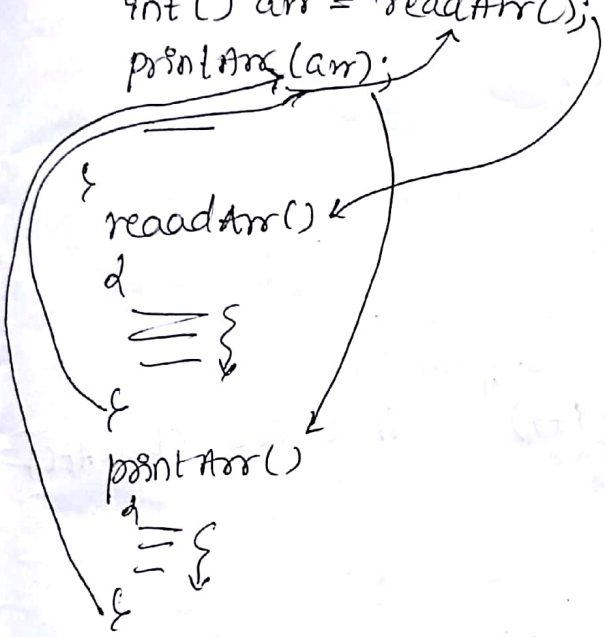
05/09/17

* Program to read array elements from user using method.

```
Import java.util.Scanner;
Class program2
{
  P S void m(String[] args)
  {
    S O P ("main method started");
    int[] arr = readArr(); // method call.
    printArr(arr); // passing array to the method.
    S.O.p ("main method ended");
  }
  P S int[] readArr()
  {
    Scanner sc = new Scanner(System.in);
    S.O.p ("Enter the size of an array");
    int size = sc.nextInt();
    int[] intArr = new int[size];
    S.O.p ("Enter the elements of array");
    for(int i=0; i<size; i++)
    {
      intArr[i] = sc.nextInt();
    }
    return intArr(); // returning array from method
  }
  P S V printArr(int[] arr)
  {
    S.O.p ("Array elements are");
    for(int i=0; i<arr.length; i++)
    {
      S.O.p(arr[i]);
    }
  }
}
```

flow of program

```
main()
{
    int[] arr = readArr();
    printArr(arr);
}
readArr()
{
    ==={
    ==}
}
printArr()
{
    ==={
    ==}
}
```



* Read an array elements from the user and find the sum of it.

Algorithm:-

Step1 : Start

Step2: read size and elements

Step3: find the sum of array elements

i.e,
```
Sum ← 0
for i ← 0 to size
Sum = Sum + a[i]
end for
print Sum.
```

program

Same program as previous except print

```
P S V SumOfArr(int[] intArr)
{
    int sum=0;
for(int i=0; i<intArr.length; i++)
  {
    sum= sum+intArr[i];
  }
```

```
        S.O.p ("Sum=" +Sum);
    }
}
```

* program to print characters

method 1:

```
String S = "JSpiders";
S.length;   // 8
sop(s);     // JSpiders

for (int i=0; i< s.lengthc(); i++)      // using charAt()
  {
    SOp(s.charAt(i));
  }
```

method 2:

```
Char[] ch= s.toCharArray()      // using toCharArray()
for (int i=0; i< ch.length; i++)
  {
    SOP(ch[i]);
  }
```

o/p:

J
S
P
i
d
e
r
s.

* Read a string from the user & check for a specific character existence. If exists, display the proper msg.

Algorithm

Step 1: Start
Step 2: Read string
Step 3: Read char to be searched
Step 4: for i ← 0 to length
        if (ch == S.charAt(i))
        {

print yes if found
return
}
end for
print not found.

program

```
import java.util.Scanner;
class program
{
  PSVM (String[] args)
  {
    SOP (" m m s");
    String s = readString(); // method call
    Char c = readChar(); // method call
    searchForChar (s, c);   // method call.
    SOP (" m m e");
  }

  PS Char readChar()
  {
    Scanner sc = new Scanner (System.in);
    System.out.println (" Enter the char to be searched");
    Char c = sc.next().charAt(0);
    return c;
  }
  PS void searchForChar (String str, Char ch)
  {
    for (int i = 0; i < str.length(); i++)
    {
      if (ch == str.charAt(i))
      {
        S.O.p (" Char "+ch + " found in string");
        return;
      }
    }
    S.O.p (" char" +ch + " not found in string "+str);
  }
```

Assignment-3

① Read an integer array from user to find the sum of even numbers.

② Read an array elements from user and check for specific elements existence. Display appropriate message

Soln:-

```
①Import java.util.Scanner;
 class program1
 {
   P S V m(String[] args)
   {
    S O p ("main method started");
    int[] arr = readArr();
    SumOfEvenArr(arr);
    S.O.p ("main method ended");
   }

   PS int[] readArr()
   {
    Scanner sc = new Scanner(System.in);
    SO.p ("Enter the size of an array");
    int size = sc.nextInt();
    int[] intArr = new int[size];
    S.O.p ("Enter the elements of array");
    for (int i=0 ; i<size; i++)
    {
      intArr[i] = sc.nextInt();
    }
      return intArr();
  }

    P S V SumOfEvenArr(int[] arr)
    {
      int sum=0;
```

```
for (int i=0 ; i< ~~int~~ a.length ; i++)
{
    if (a[i]%2==0)
    {
        sum = sum + a[i];
    }
    S.O.p (" Sum of even numbers in array=" +sum);
}
}
```

② class Search

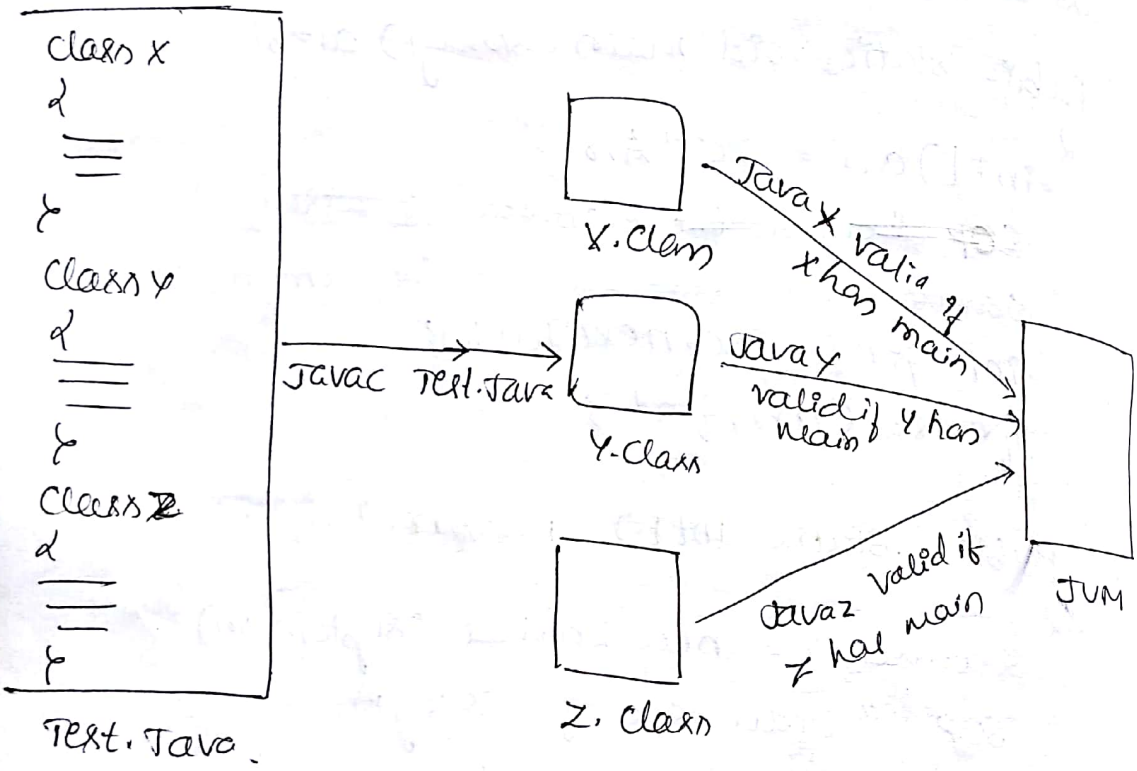
```
{
    Public static void main (String[] args)
    {
        int[] arr = readArr();
        ~~SOP (" Enter the elements of array")~~
        ~~Scanner sc = new Scanner (System.in);~~
        int find = sc.nextInt();
        ifExists (arr, find);
    }
    Public .Static int[] readArr()
    {
        Scanner sc = new Scanner (System.in)
        SOP (" Enter size of array");
        int size = sc.nextInt();
        int[] arr = new int[size];
        SOP (" Enter element of array");
        for (int i=0; i< size ; i++)
        {
            arr[i] = sc.nextInt();
        }
        return arr;
    }
    Public Static void ifExists (int[] arr, int find)
    {
```

```
for (int i=0; i<arr.length; i++)
  {
    if (arr[i] == find)
    SOP ("Element found");
    return;
  }
}
}
```

06/09/17

Section - II

CLASSES & OBJECTS



* No of bytecode = No of class in source file

Assignment - 4

① program to count even & odd integer in array

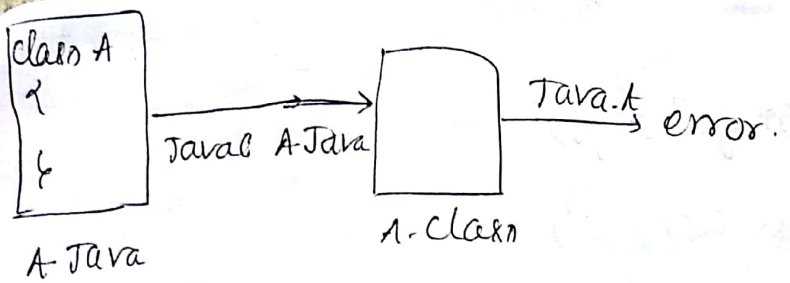
② program to check whether the given number is a palindrome or not.

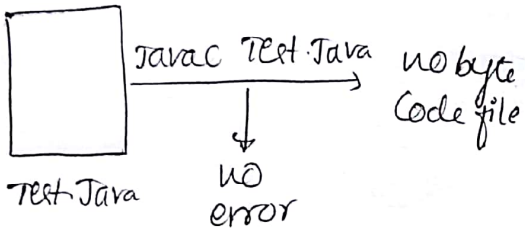
Eg:
```
class X
{
  P S V M (String[] args)
  {
    S O P ("In class X");
  }
}
class Y
{
  P S V M (String[] args)
  {
    S O P ("In class Y");
  }
}
class Z
{
  P S V M (String[] args)
  {
    S O P ("In class Z");
  }
}
```

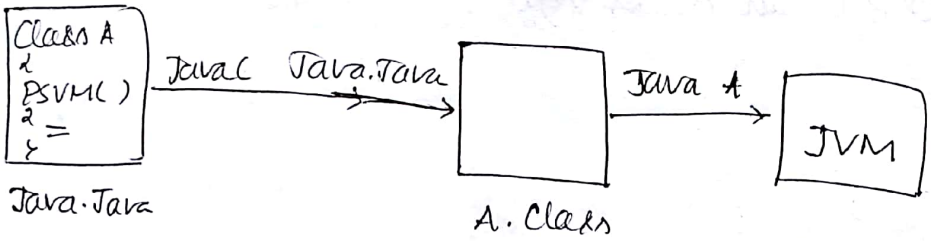
* Whenever we compile resource file, the number of bytecode file generated = number of classes present in source file.
* For the JVM to execute, we have to provide only one .class file at a time, JVM will execute that .class file only if it contains the main method according to the below syntax.

Public Static Void main (String[] args)

* If the byte code file which is given to the JVM to execute doesn't contain the main method or main method doesn't follow the above syntax then we will get run time error saying "main method not found".







Note:

* An empty class file can be compiled successfully but cannot be executed.
* An empty source file can be compiled without error but no byte code file will be generated.

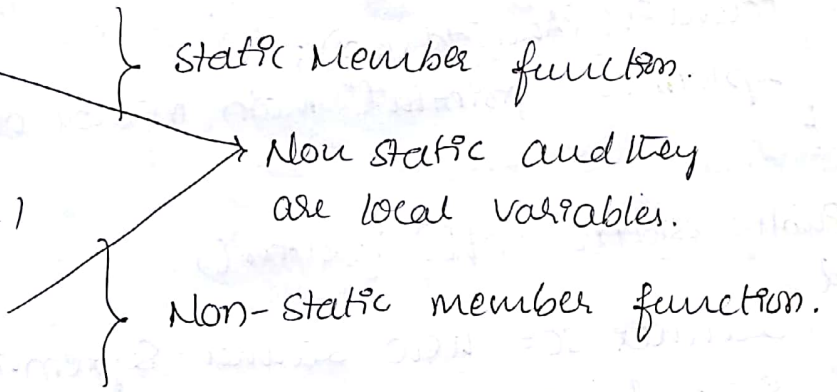
```
class className
{
    // Declare & Initialize static member
    // Declare & Initialize Non-static member (Instance)
}
```

static member → Data member, Member function

Non-static member → DM, MF

```
Class Test
{
    static int x;  ⟶ static DM
    int y;         ⟶ Non-static DM

    static void m1()
    {
        int
        ≡
    }
    void m2()
    {
        int n;
        ≡
    }
}
```



* Accessing static Members of the class.

Static Members of the class can be accessed by using the ClassName.

ClassName. Static Member Name.

```
Class A
{
    static int x=10;
    static void m1();
    {
        ≡ {
    }
}
Class program1
{
    PS Vm (__)
    {
        A.x; //10
        A.m1();
    }
}
```

Prgm1. Java.

Assignment-4

① import java.util.Scanner; → class program1

```
{
Public static void main (String[] args)
{
  System.out.println (" main method started");
  int[] arr = readArr();
  CountEvenAndOdd (arr);
  System.out.println (" main method ended");
}

Public static int[] readArr()
{
  Scanner sc = new Scanner (System.in);
  S.O.P (" Enter the size of an array");
  int size = sc.nextInt();
  int[] intArr = new int[size];
  S.O.P(" Enter the elements of array");
  for (int i=0; i<size; i++)
    {
     intArr[i] = sc.nextInt();
    }
     return intArr();
}

Public static void CountofOddEven (int[] intArr)
{
  int Ocount = 0;
  int Ecount = 0;
  for (int i = 0; i < intArr.length; i++)
  {
    if (intArr[i] %2 == 0)
    {
     Ecount = Ecount+1;
    else
    { Ocount = Ocount+1;
    }
    S.O.P (" Sum of even numbers is " + Ecount + " and
}}          " Sum of odd numbers is " + Ocount);
```

(2) class palindrome → Class program2.

```
{
public static void main (String[] args)
{
System.out.println("Main method started");
Scanner sc = new Scanner(System.in);
System.out.println("Enter the number");
int Palindrome = sc.nextInt();
isPalindrome(palindrome);
System.out.println("Main Method Ended");
}
Public static void isPalindrome (int Pal)
{
int num = pal;
int rev = 0;
while(num!=0)
{
int rem = num%10;
rev = rev*10+rem;
num = num/10;
}
if(pal == rev)
{
System.out.println("Number "+pal+" is a
palindrome");
}
else
{
System.out.println("Number "+pal+" is not
a palindrome");
}
}
}
```

07/09/17

* Accessing the static members of class.

Eg:
```
class A
{
  static int x=10;
  static void m1():
  {
    S.O.P ("Static method of class");
  }
}
class B
{
  static double x= 54.54;
  static void m1()
  {
    S.O.P (" static method of B class");
  }
}
class programs
{
  P S V m(String[] args)
  {
    S.O.P ("Main Method started");
    S.O.P (" A.x = " +A.x);
    A.m1();
    S.O.P (" B.x = " +B.x);
    B.m1();
    S.O.P (" Main method ended");
  }
}
```

* Using class name we access static members
* Static and Non-static data members by default could the default value.
* Static members can be reassigned any number of ti (as long as it is not a Final)
* We need not initialize variable to '0' in case of static & Non static variable. Only for local z=0.

```
class A
{
    static int x = 10;
    static final double y = 30.32 // can not re-assign
    static int z;
}
class pgm2
{
    P S V m (String[] args)
    {
        S.O.P("main method started");
        S.O.P("Before change");
        S.O.P("A.x = " + A.x); //10
        A.x = 200;                    // reassignment
        S.O.P("After changed");
        S.O.P("A.x = " + A.x); // 200
        S.O.P("A.y = " + A.y); // 30.32
    // A.y = 432.43; // Error, bcz y is final
        S.O.P("A.z = " + A.z); // 0
        S.O.P("Main method ended");
    }
}
```

O/P:
```
MM Started
Before change
A.x = 10
After change
A.x = 200
A.y = 30.32
A.z = 0
M-M ended.
```

* Accessing Non-static members ~~Accessing~~

→ Non-static member of the class can be accessed by using the following 2 steps.

(a) Create an object of a class by using new operator or new keyboard.

```
new ClassName();
```

Eg: new Demo(); // Object of Demo class
new Test(); // — " — Test Class
new Sample(); // — " — Sample Class.

→ Create a Reference variable and make it to point to created object.

```
Type RefVarName = ;
```

Class type — Interface type.

Eg: A a ; // It is Class type (or) a is of type A
Test t ; // t is type Test
Sample s ; // s is Sample type

* Reference variable:

→ It is used to refer an object, using the reference variable we can access the members of the refered object

→ Reference variable can be a class type or an interface type or enum type.

→ For the reference variable either we can assign an obj or NULL (we cannot assign primitive value to reference variable)

```
Class A
{
    int x = 10;
    void m1().
{
```

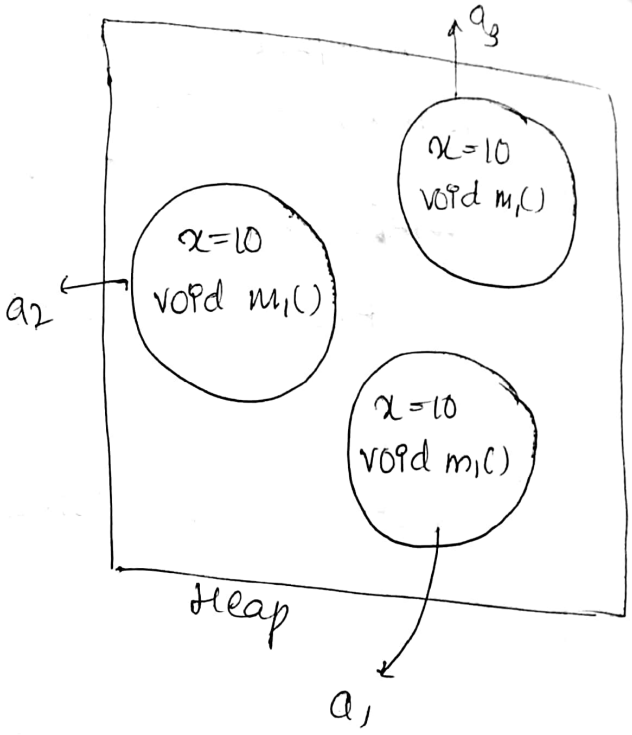
A a_1, a_2, a_3 ; // Ref. var of class type

A a_4;

a_1 = new A();

a_2 = new A();

a_3 = new A();



$a_1.x$
$a_2.x$ } 10
$a_3.x$

$a_1.m_1()$;
$a_2.m_2()$; } ∪
$a_3.m_3()$;

$a_1.y$;
$a_2.y$; } Error
$a_3.y$;

a_4 = null

$a_4.x$; // Exp: nullpointer Exp.

Assignment - 5

① Program to read integer number from the user & check whether it is prime or not

② Program to read integer number from the user & check whether it is perfect number.

soln

① import java.util.Scanner;

{

Public static void main (String[] args)

{

System.out.println("main method started");

~~int[] arr = readArr();~~

Scanner sc = new Scanner(System.in);

S.O.P ("Enter the number");

int num = sc.nextInt();

isprime(num);

S.O.P("main method ended");

```
Public static void isPrime(int number)
{
    int m = num/2, count=0;
    for (int i=2; i<=m; i++)
    {                                   if (m%i ==0)
      if (m%i ==0)                      { count++;
      {                                   break;}
        S.O.P (+m " is prime");
      }
      else
      {
        S.O.P(+m " is not Prime");
      }
    }
}
```

count

② import java.util.Scanner; → class prgrm1{

```
Public static void main (String[] args)
{
 S.O.P ("main method started");
 Scanner sc = new Scanner (System.in);
 S.O.P (" Enter the number");
 int num = sc.nextInt();
  perfect (num);
 S.O.P(" main method ended");
}
Public static void ~~main~~perfect (int num).
{
  int sum=0;
  for (int i=1; i<num; i++)
  {
    if (num % i == 0)
    {
      sum = sum +i;
    }
  }
  if (sum == num)
  {
    S.O.P (" Given number is perfect");
  }else
```

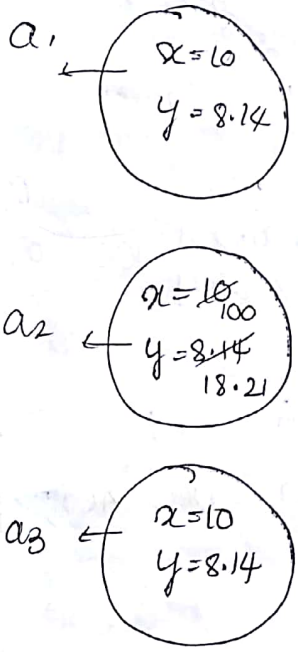
```
    S.O.P(" Given number is not perfect");
  }
}
```

08/09/17

Reference variable.

Eg: ①

```
class A
{
  int x=10;
  double y=8.14;
}
A a1, a2, a3; // Ref variable of type A
a1 = new A();
a2 = new A();
a3 = new A();
      :
```



pointing to different object.

a1.x, a2.x, a3.x } 10

a1.y, a2.y } 8.14, a3.y

a2.x = 100;

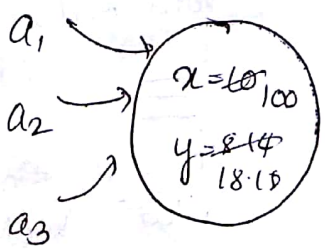
a2.y = 18.21;

a1.x, a2.x } 10, a2.x ⟹ 100

a1.y, a3.y } 8.14, a2.y; // 18.21.

Eg ②

```
class A
{
  int x=10;
  double y=8.14;
}
A a1, a2, a3; // Ref var of type A
a1 = new A();
a2 = a1;
a3 = a2;
int x = 10 //
```



y = x;

$a_1.x$;
$a_2.x$; } 10
$a_3.x$

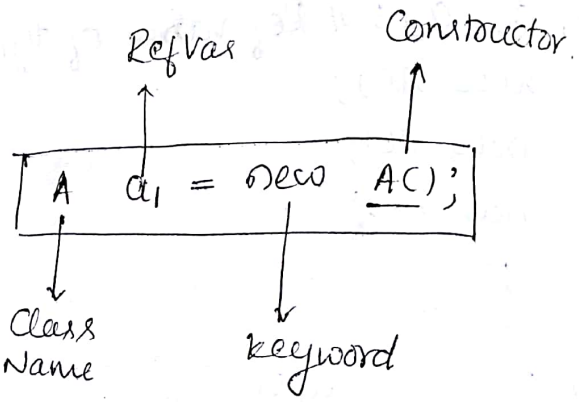
$a_1.y$
$a_2.y$ } 8.14
$a_3.y$

$a_1.x = 100$
$a_2.y = 18.18$

$a_2.x$
$a_1.x$ } 100
$a_3.x$

$a_1.y$
$a_2.y$ } 18.18
$a_3.y$

A a_1;
a_1 = new AC(); }



* static members of the class belongs to the entire whereas non-static members of the class belongs to individual object

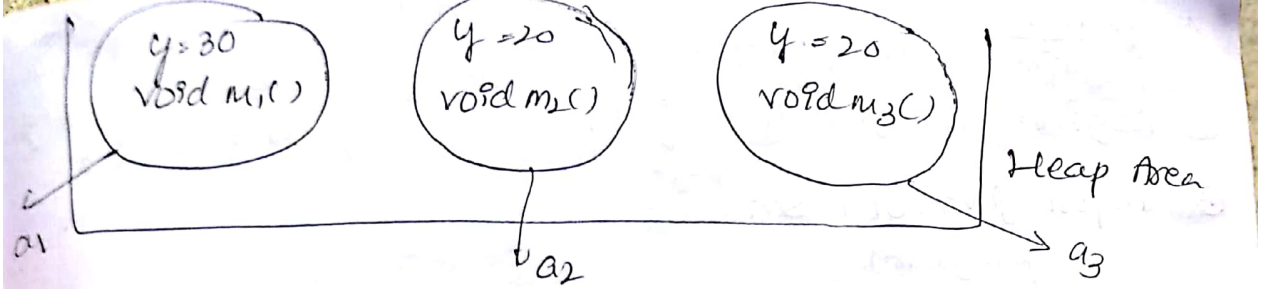
* static members will be loaded into the memory at the time of class loading. Whereas non-static me will be loaded into the memory at the time of object creation.

Eg: class A
{
static int x=10;
int y=20;

void m1()
{
=
=
}
static void m2()

A a_1 = new A();
A a_2 = new A();
A a_3 = new A();

void m2() x 10



* If we want to represent any common information across multiple people, then we should declare that common information as static. So that, that common information is maintained at class level instead of object level.

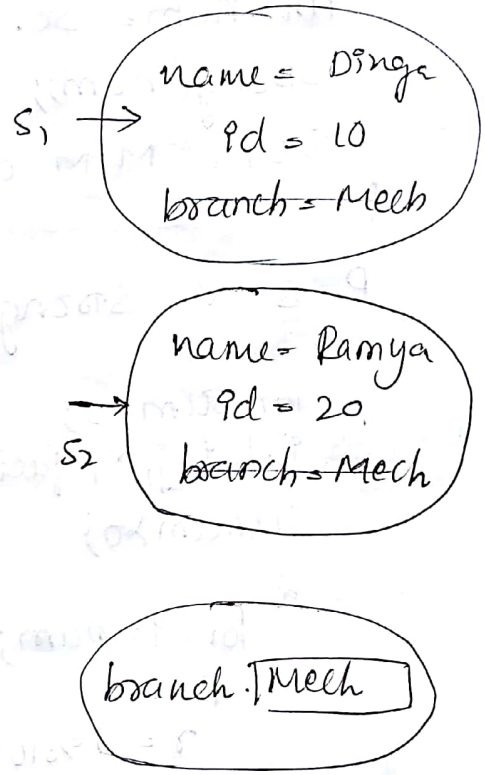
Eg: class Student
{
String name;
int id;
static String branch;
}
Student s1 = new student();
s1.name = "Dinga";
s1.id = 10;
~~s1.branch = "mech";~~

Student s2 = new student();
s2.name = "Ramya";
s2.id = 20;
~~s2.branch = "mech";~~

⋮

50 Students

Student.branch = "Mech"



* INSTANTIATION

The process of creating an object of a class is known as Instantiation. i.e., If we want to access non-static members of the class we should instantiate the class

Eg: class A
{
int x=10;
}
new A().x //10

A a = new A();

Assignment-6

① Program to check whether the given number is Strong

② Program to find the factorial of a given number.

① Import java.util.Scanner

```
class program1
{
  P.S.V. main(String[] args)
  {
    S.O.P("M.M Started");
    Scanner sc= new Scanner(System.in);
    S.O.P("Enter a number");
    int num= sc.nextInt();
    Strong(num);
    S.O.P("M.M. Ended");
  }

  P.S.V Strong(int num);
  {
    int sum=0;
    int i, j, r, fact;
    if(num>0)
    {
      for(i=num; i!=0; i=i/10)
      {
        r= i%10;
        fact=1;
        for(j=1; j<=r; j++)
          fact= fact*j;
        sum= sum+fact;
      }
      if(num==sum)
        S.O.P("number is Strong");
      else
        S.O.P("number is not Strong");
    }
    else
      S.O.P("num is not Strong");
  }
```

(2)
```
Import java.util.Scanner
class factorial
{
  P.S.V. main (String[] args)
  {
    S.O.P ("main method started");
    Scanner sc = new Scanner (System.in);
    S.O.P ("Enter a number");
    int num = sc.nextInt();
    fact (num);
    S.O.P ("main method ended");
  }
P.S.V fact (int num)
  {
    int factor = 1, n, c;
    if (n < 0)
    S.O.P ("Number should be Non-negative");
    else
    {
      for (c = 1 ; c <= n; c++)
      {
        factor = factor * c;
      }
    S.O.P ("factorial of " +num " is = " + fact);
  }
}
```

18/09/17

*Object Initialization

The process of initializing the members of object is known as object initialization.

⇒ Object can be initialized used 2 blocks

① Instance block (Rarely used)

② Constructor

* Blocks:

→ These are used to execute certain instructions.

→ Blocks are classified into 2 types

① static block ② Instance block (non-static)

Eg 1:

```
class Test
{
    {
      S.O.P ("Inside non-static block");
    }
    static
    {
     S.O.P ("Inside static block");
    }
}
class prgm1
{
    P.S.V. main (String[].args)
    {
      S.O.P ("Main method started");
      new Test();
      S.O.P ("Main method ended");
    }
}
```

O/P:

```
Main method started
Inside static block
Inside Non-static block
Main method ended
```

```
class Test
{ static void m1();
  {
  }
  void m2()
  {
  }
  static
  {
   // Static Block
  }
  {
   // Instance Block
  }
}
```

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

(i) Create an obj in heap
(ii) Load all Non-static members into the obj
(iii) Execute static block (if exists)
(iv) Execute Non-static (if exists)
(v) Execute Constructor

Usually Non-static exists.

* Initializing objects through instance block

```
Eg: class Test
    {
      int x;
      int y;
      {
        x=10;
        y=32.32;
      }
    }
    class program2
    {
      P.S.V. main (String[] args)
      {
        S.O.P("M M S");
        Test t = new Test();
        S.O.P("t.x = " +t.x);
        S.O.P(" t.y = " +t.y);
        S.O.P(" M.M.E ");
      }
    }
```

* CONSTRUCTOR:

→ Special member functions of the which will have the same name as that of the class.

→ Called automatically, when object is created. and hence they donot have any return type.

→ These are generally used to initialize an obj

SYNTAX:

```
Access Modifier  ClassName()
{                Public / private / protefor / default
    ≡≡≡   } Constructor body.
}
```

→ Constructor can be defined either by the user or by the compiler.

→ User can define both No-arg & argument Constructor (parameterised Constructor). Whereas compiler will define only no-argument constructor.

→ Compiler defined no-arg Constructor is also known as default Constructor

→ Every java program will have either user defined or compiler defined constructor. but not both.

Eg:

```
class A
{
    int x;
    double y;
    A()
    {
        x = 10;
        y = 8.14;
    }
}
```

used defined No-arg Constructor

```
class B
{
    int x;
    double y;
}
```

In this class default Constructor exists.

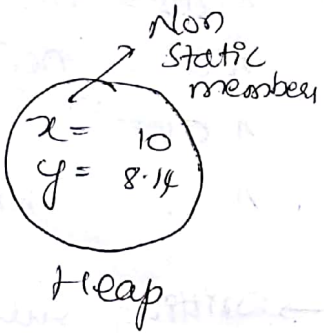
Note:

All default constructors are No-arg but All No-arg Constructor are default constructors //

Eg:
```
class A
{
  int x;
  double y;
  A()
  {
    x=10;
    y=8.14;
  }
}
```

A a = new A();

a.x //10
a.y //8.14

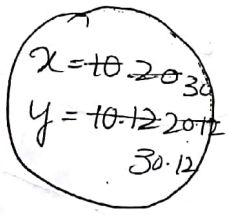


Note: constructors are not used to create an object. Instead constructors will be executed in the process of creating an object.

Eg:
```
class A
{
  int x=10;
  double y=10.12;
  {
    x=20;
    y=20.12;
  }
  A();
  {
    x=30;
    y=30.12;
  }
}
```

A a = new A();

a.x // 30
a.y //30.12



* Arguement Constructor.

→ Also known as parameterised Constructor.
→ If we want multiple objects to have different values then we'll go for arguement constructor.
→ Eg:

```
class A
{
  int x;
double ~~int~~ y;
  A(int arg1, double arg2)
  {
    x=arg1;
    y=arg2;
  }
}
```

2 arg ctor defined.

A a1 = new A(10, 10.12); ← x=10, y=10.12

A a2 = new A(20, 20.12); ← x=20, y=20.12

A a3 = new A();

A a4 = new A(100);

A a5 = new A(1, 2, 3);

} Error bcz vary apart from 2 argument.

→ Writing only argument doesnot allow us to create an object with different length and type of argument.

→ To create an object with different type of argument and different length of argument then we should go for Constructor Overloading.

19/09/17

* Constructor Overloading.

Eg:

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

Constructor overloading

A a1 = new A(); a1 → x=0, y=0.0

A a2 = new A(100); a2 → x=100, y=0.0

A a3 = new A(8.14);

A a4 = new A(10, 88.11);

A a5 = new A(10,20,30); ✗ Error.

all are integer type and 3 argument.

→ What is constructor overloading?

Writing a constructor with different length of argument and different type of argument is known as C.O.

When we overload a constructor in a program, compiler will call respective constructor by looking at number of argument or type of argument or order of occurance of argument.

→ Advantages:

Constructor overloading helps us in creating an object with different number of ~~objects~~ argument and different type ~~of~~ argument

→ Conclusion 1

Constructors donot have any return type. If we specify any return type to a constructor then constructor will be converted into method, which needs to be called explicitly by creating an object.

Eg: class A
{
void A()
{
S.O.P("Hello");
}
}
A a = new A();
a.A();

→ Conclusion 2

Eg: class A
{
// default constructor exists, if is not empty.
}

→ Conclusion 3

Eg: For a constructor we can specify any access modifier. However default access modifier of constructor is same as that of the class.

→ Conclusion - 4 (Constructor called during object creation.

Constructor cannot be declared as static. Bcz Constructor will be executed at the time of object creation.

→ Conclusion - 5

Eg: Class A
{
A()
{
}
A(int arg)
{
}
A(double arg)
{
}
}

new A();
new A(10);
new A(8.14);
new A('A');
new A(8.18f);
new A(9999L);

map goes forward

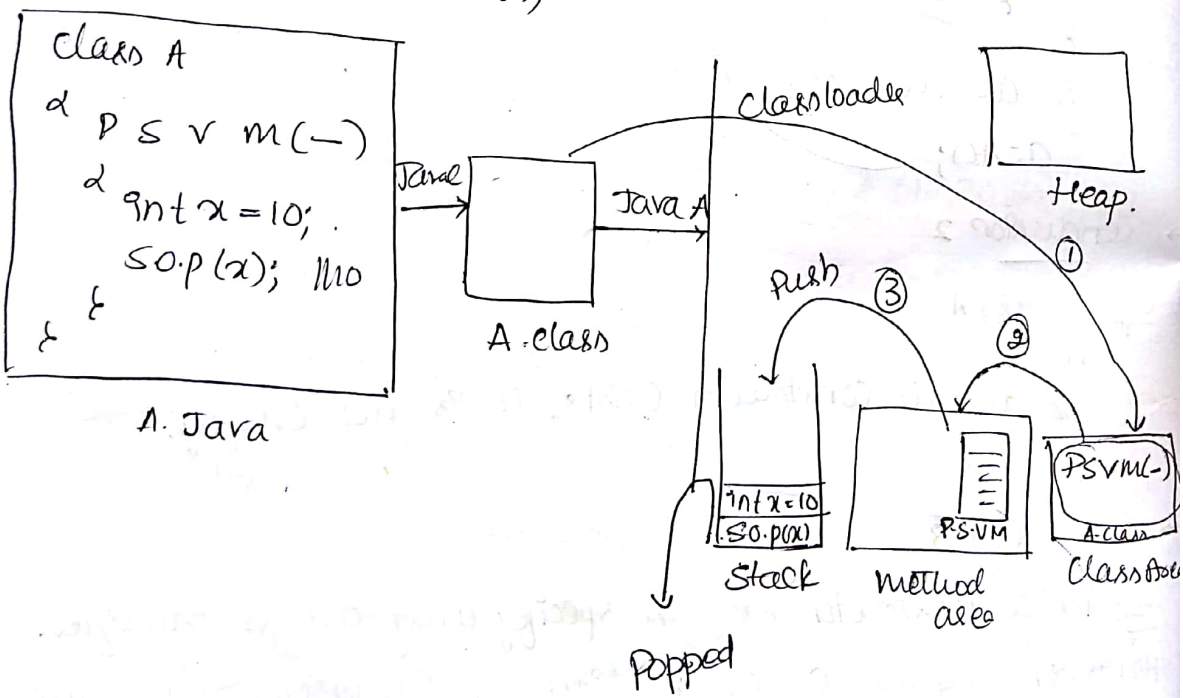
byte → short
char → int → long → float → double

Since no char given to int

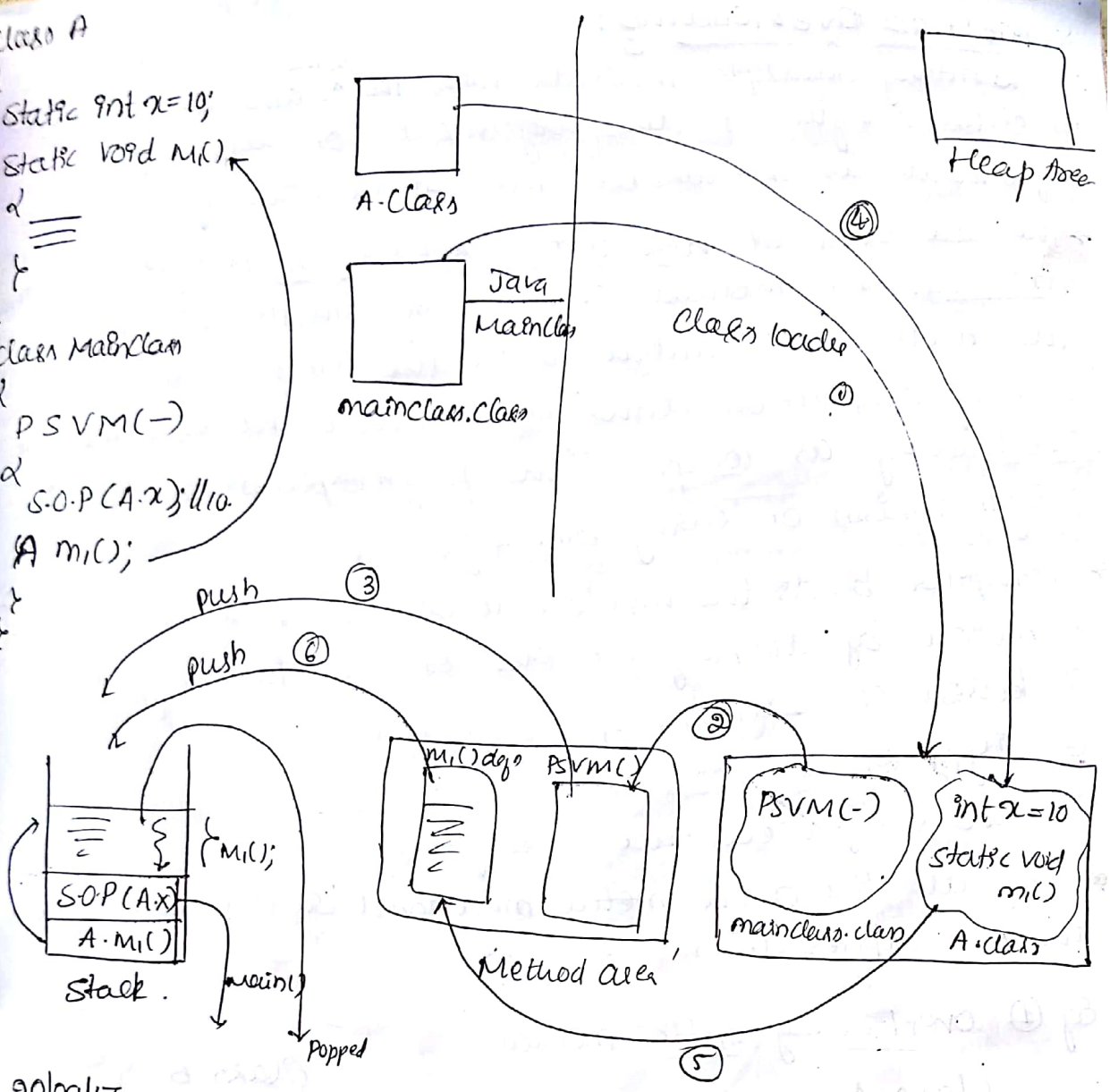
Static loaded during before execution.

* JVM memory.

→ Heap Area (objects)
→ Class Area (static info)
→ Method Area (Def" of static & Non-static method)
→ Stack Area (Execution)

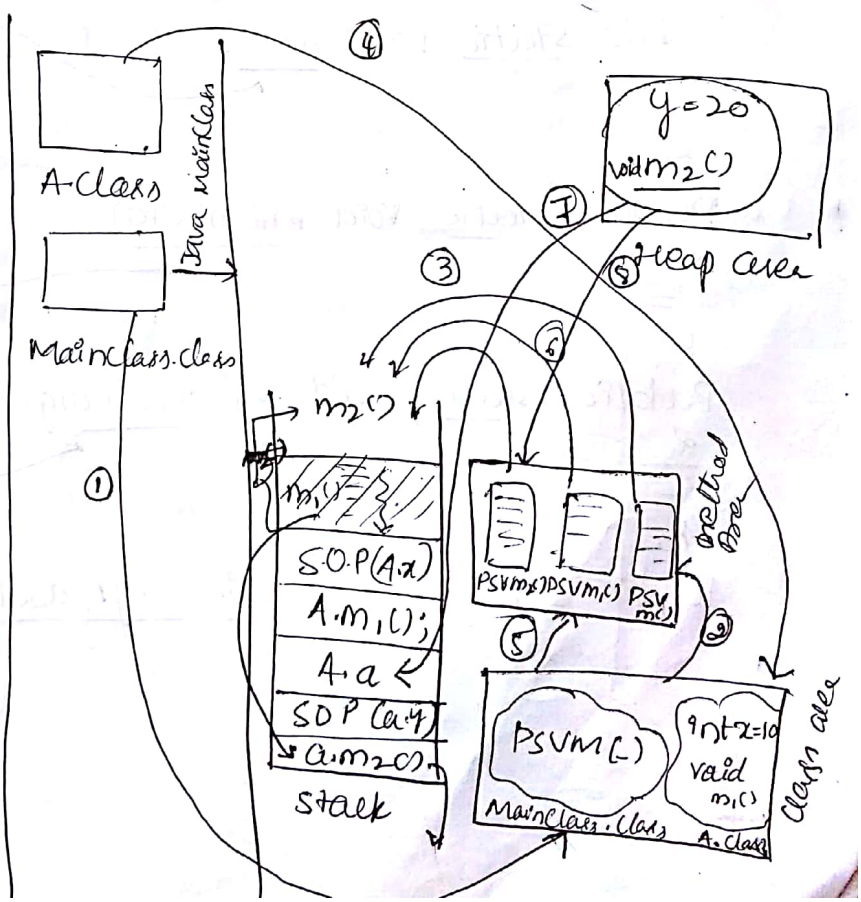


```
class A
{
  static int x=10;
  static void M1()
  {
  }
}
class MainClass
{
  PSVM(-)
  {
    S.O.P(A.x);//10.
    A m1();
  }
}
```



20/09/17

```
class A
{
  static int x=10;
        int y=20;
  static void M1()
  {
  }
  void m2()
  {
  }
class MainClass
{
  PSVM(-)
  {
    S.O.P(A.x);//10
    A.M1();
    A a = new A();
    S.O.P(a.y);//20
```



* METHOD Overloading:

Writing multiple methods with the same name but different in either length of the argument or type of the argument is known as Method overloading.

→ We can overload both static as well as not-static methods. In method overloading compiler will bind the method declaration with the method defination during compilation time and hence we call method overloading as compile time polymorphism or static polymorphism or early binding.

→ Compiler binds the method declaration with method defination by looking at one of the following criteria

① Based on number of arguments
② Type of arguments.
③ Order of occurance of argument.

* If all the above mentioned criteria are same then compiler throw error.

Eg ① 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)
  {
  }
}
```

```
Class B
{
  PSVM()
  {
    A.m1();
    A.m1(10);
    A.m1(8.14);
    A.m1(10,8.14);
  }
}
```

② write a program to demonstrate non static method overloading.

```
class A
{
 public void m1()                    A.a = new m1();
 {
 }
 public void m1(int arg)             A a = new m1(10);
 {
 }
 public void m1(double arg)          A a = new m1(8.14);
 {
 }
 public void m1(int arg1, double arg2)   A a = new m1(10,8);
 {
 }
```

* Overloading Main method:

Eg:

```
class A
{
 p s v m()
 {
  S.O.P("no arg main method");
 }
 p s v m(int arg)
 {
  S.O.P("int arg main method");
 }
 p s v m(double arg)
 {
  S.O.P("double arg main method");
 }
 p s v m(int arg1, double arg2)
 {
  S.O.P("int and double main method");
 }
}
class program1
{
 p s v m(String[] args)
 {
```

```
        S.O.P ("main method started");
        Test.main();
        Test.main(100);
        Test.main(323.23);
        Test.main(10, 323.23);
        S.O.P ("Main method ended");
    }
}
```

* Advantage of Overloading

→ Reduces the complexity of the user.

Eg: Without overloading

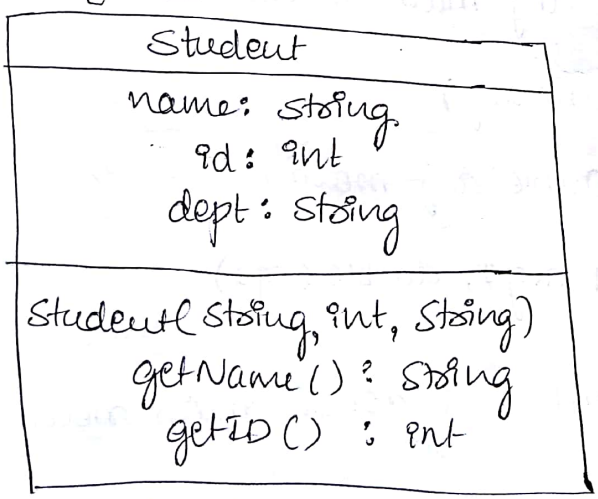
```
void isort(int[] arr)
{
// logic to sort int number
}
void dsort(double[] arr)
{
// logic to sort double num
}
void csort(char[] ch)
{
// logic to sort char
}
```

With overloading.

```
void sort(int[] arr)
{
// logic to sort int num
}
void sort(double[] arr)
{
// logic to sort double
}
void sort(char[] ch)
{
// logic to sort char
}
```

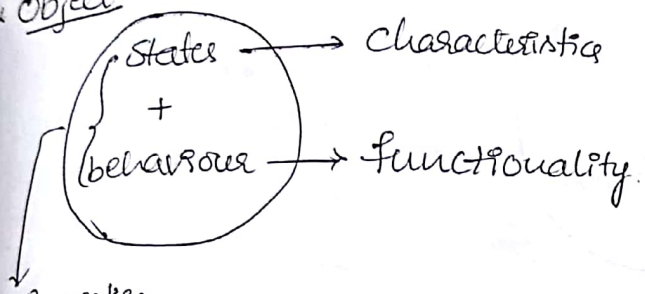
21/09/17

* Class Diagram:-



class diagram.

Object



An entity which has its own state and behaviour is known as object.
States represents characteristics of object. Whereas behaviour represents functionality of an object

```
class car
{
    String name;
    double price;
    String color;
    void commute()
    {
    }
}
```

```
Car C1 = new Car();
C1.name = "Honda";
C1.price = 1200000.00;
C1.color = "Black";

Car C2 = new Car();
C2.name = "Audi";
C2.Price = "50,00,00.00;
C2.color = "Red";
```

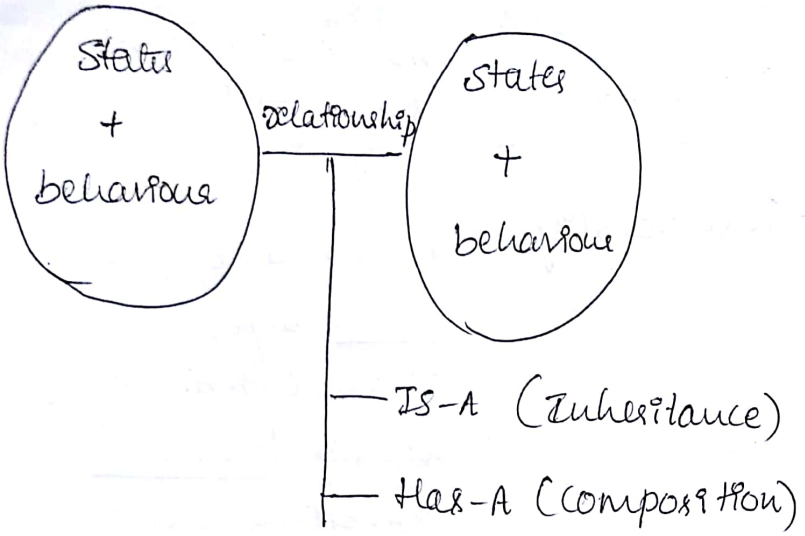
C1 → name: Honda, price: 1200000.00, Color: black, void commute()

C2 → name = Audi, price = 500000.00, color = Red, void commute()

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

In class States represents the data members of the class. Whereas Behaviour represents the member function of the class.

Whenever we create multiple objects of a class all objects will have same states and behaviour but different values.



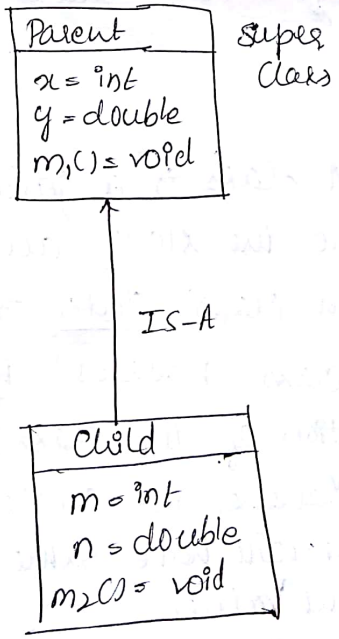
* INHERITANCE [IS-A]

→ It is a process of deriving the properties of one class from another class.
→ The class from where the properties are inherited is known as Parent class / Base class / super class.
→ The class to which the properties are inherited is called child class / derived class / sub class.
→ We can establish the relationship between 2 class by using the keyword extends.

* Types of Inheritance :-

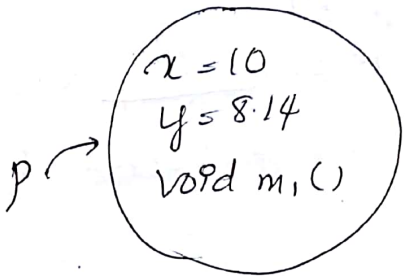
① Single level Inheritance - In this type of inheritance we will have a single super class and single sub-class.

Eg: class parent
{
int x=10;
double y= 8.14;
void m1()
{
}
}
class child extends parent
(subclass) (keyword) (super class)
{
int m=20;
double n=11.12;
void m2()
{
}



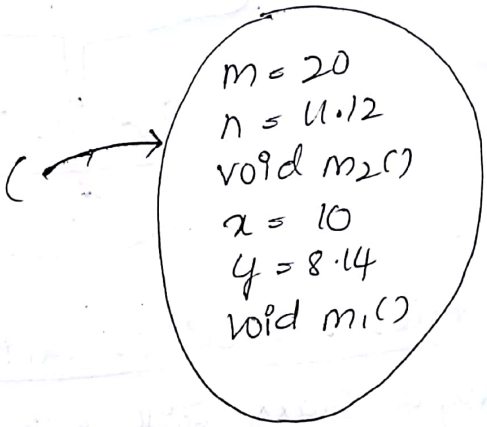
parent p = new parent();

p.x ✓
p.y ✓
p.m1() ✓
p.m ✗
p.n ✗
p.m2() ✗



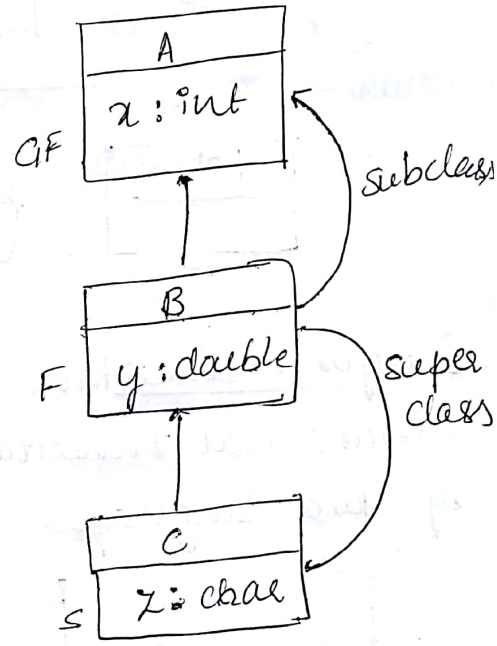
Child c = new Child();

c.m ✓
c.n ✓
c.m2() ✓
c.x ✓
c.y ✓
c.m1() ✓



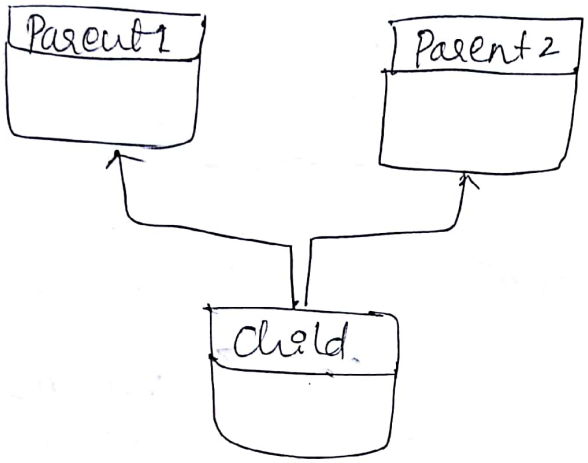
② Multi-level Inheritance :-

Eg:
```
class A
{
  int x=10;
}
class B extends A
{
  double y=8.14;
}
class C extends B
{
  char z='A';
}
```



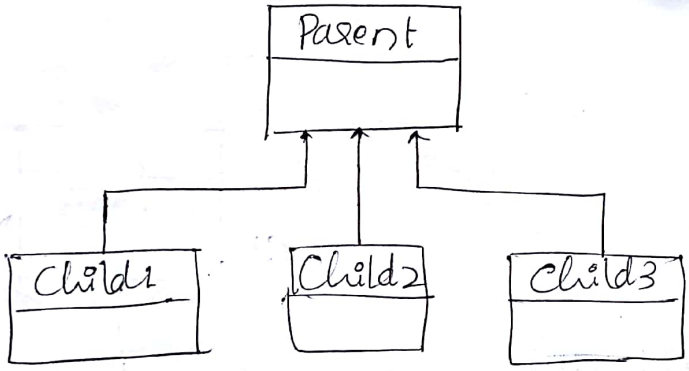
26/09/17

③ Multiple Inheritance: In this type of inheritance sub class will have more than 1 super classes.

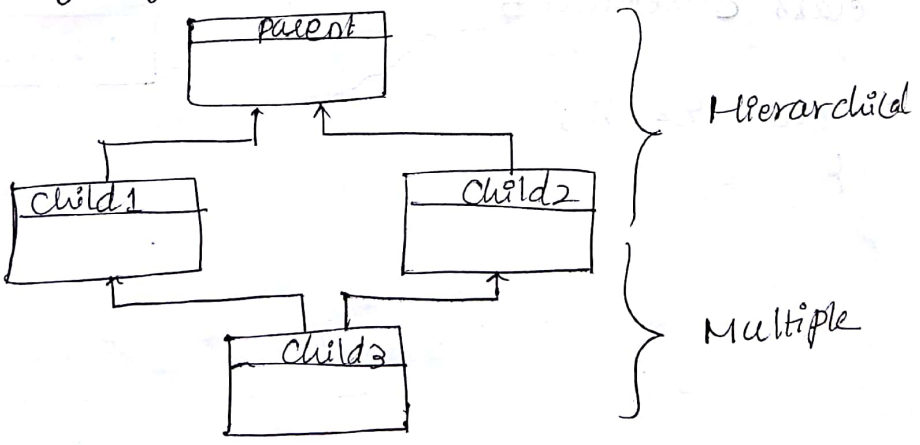


✗ Not allowed in Java.

④ Hierarchical Inheritance: In this type of inheritance a super class will have multiple subclass. This is one of the most widly used inheritance in industry.



⑤ Hybrid Inheritance: Combination of multiple and hierarchical Inheritance. This is also not supported by Java language.



Conclusions

① In java every class is a subclass of object class. In other words Object class acts as super most class in the entire java hierarchy.

② Object class available in java.lang pkg.

Eg: ① class A extends Object → compiler writes it.
{
}

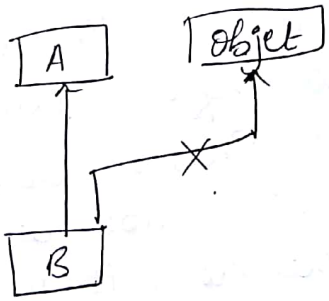
② class B
{
// It contains default constructor ⓣ
// properties of object class.
}

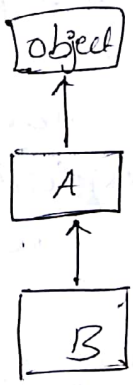
③ If a class is declared as final, we cannot inherit from such class. In other words, A final class will not have any subclass. (They do have super class).

Eg: final class A
{
}
class B ~~extends A~~
{
}
→ Error.

* String
* String buffer } Can't inherit.
* String builder

④ class A extends Object
{
}
class B extends A
{
}

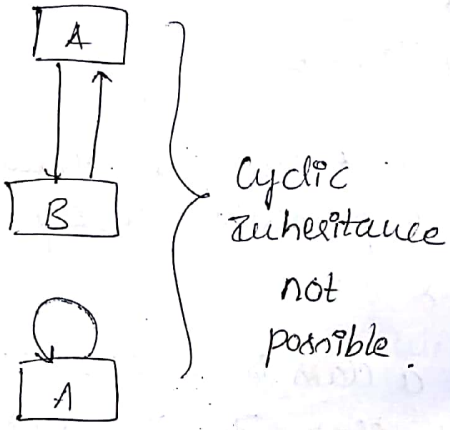




④ Whenever we inherit a sub class from super class, all properties of super class will be inherited to subclass. Except the following 3 properties.

ⓐ private members
ⓑ Constructors
ⓒ static members

⑤
```
Class A extends B
{
==
}
Class B extends A
{
==
}
Class A extends A
```



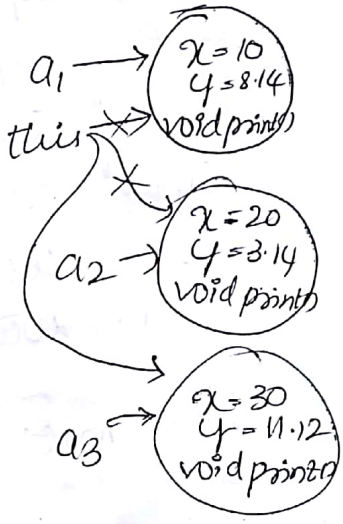
* Advantages of Inheritance.
→ Code Reusability.
→ Easy to enhance the application.

* THIS keyword.
→ keyword used to refer the current object. An object through which we are invoking a method is known as current object.
→ This keyword will point to only one object at a time.
→ When the local variable names are same as instance variable names, then we should compulsorily use this explicitly to refer instance member. Otherwise priority will be given to local variable.
→ This keyword cannot be used inside a static method body. In other words, it must be used inside a non-static method body or constructor.

Eg: class A
{
int x;
double y;
A (int x, double y) ← local variable
{
this.x = x;
this.y = y;
}
void print()
{
int x = 100; } local
int y = 10.11;
SOP(x); // this.x
SOP(y); // this.y
compiler does only when there is no local variable
S.OP(this.x);
SOP (this.y);
}
}

A a1 = new A(10, 8.14);
A a2 = new A(20, 3.14);
A a3 = new A(30, 11.12);

a1.print(); //10, 8.14;
a2.print(); //20, 3.14
a3.print(); // 30, 11.12



21|09|17

* SUPER keyword.

→ It is a keyword used to refer super class members from the sub class, when super class members and sub class member names are same.

→ Like This keyword, super keyword should also be used either within non-static method body or within constructor.

Eg: class parent
{
int m=10;
double n = 8.14;
void M1()
{
S.O.P(" Parent M1() method ");
}
}

```
Class child extends parent
{
  int m= 200;
double n = 3.14;
void print()
{
  int m=100;
double n= 434.23;
  S.OP("super class m= "+super.m);
  SOP(" super class n= "+super. n);
  s.O.PL" sub class  m= "+sub this.m);
  S.O.PL" sub class  n= " + this.n);
  S.O.P(" local m = "+m);
  S.O.PL" local n =  "+n);
}
 void m1()
 {
  super. m1();
  }
}
Class prgm2
 {
    P. S V  m (String[] args)
    {
       S.O.P(" Main method started");
        child c=new child();
            c. print();
            c. m1();
       S.O.P (" Main method ended");
     }
}
```

* CONSTRACTOR CALLING:

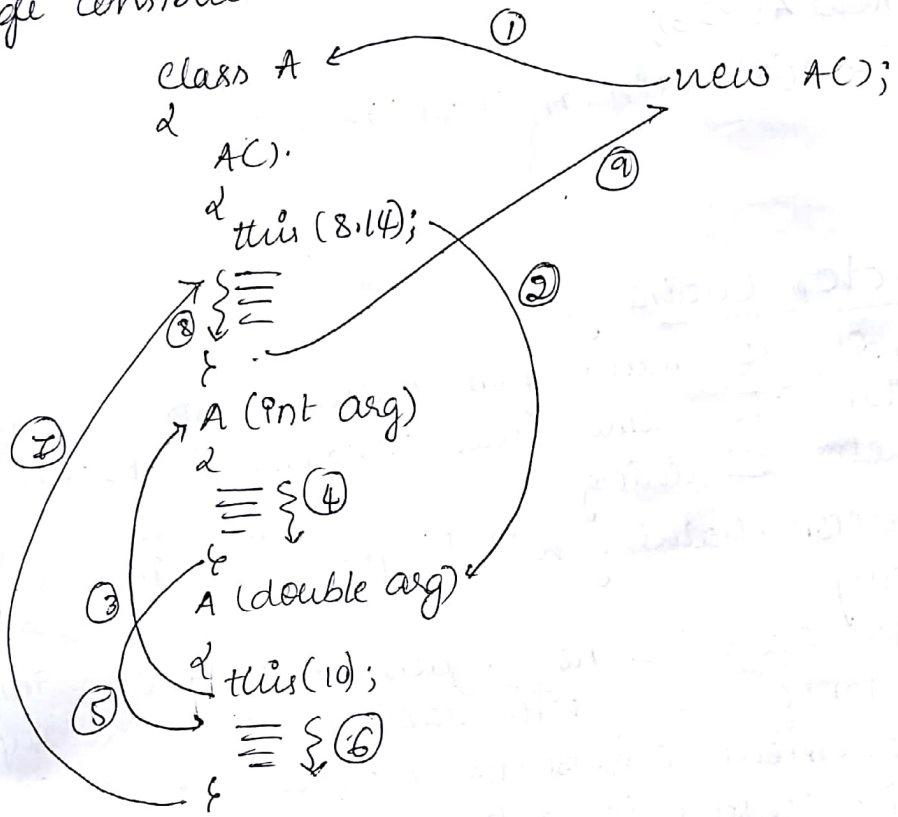
→ process of calling one constructor from another constructor of the same class.

→ Once we call one constructor for one object, we cannot call same constructor for same object.

i.e., Recursive constructor calling is not allowed.
(Recursive function calling is allowed)

→ constructor calling statement should be the first statement inside the constructor body and hence we cannot call more than one constructor within a single constructor.

Eg: ①



②
```
class A
{
  A()
  {
    this (32);
    S.o.p(" Inside no-arg constructor");
    // this (321); Error bcz it should be I statement
  }
  A (int arg)
  {
    this ( 3.32);
    S.o.P (" Inside int-arg constructor");
  }
```

```
A (double arg)
{
  // this(); Error bcz recursive constructor calling not allowed.
  S.O.P (" Inside double-arg constructor");
}
}
Class pogrm3
{
  P S V m (String[] args)
  {
    S.O.P ("main method started");
    new A();
    new A(323);
    S.O.P("Main method ended");
  }
}
```

* Constrctor Chaining:

→ Calling one constructor from another constructor, that constructor calls some other constructor is known as Constructor Chaining.

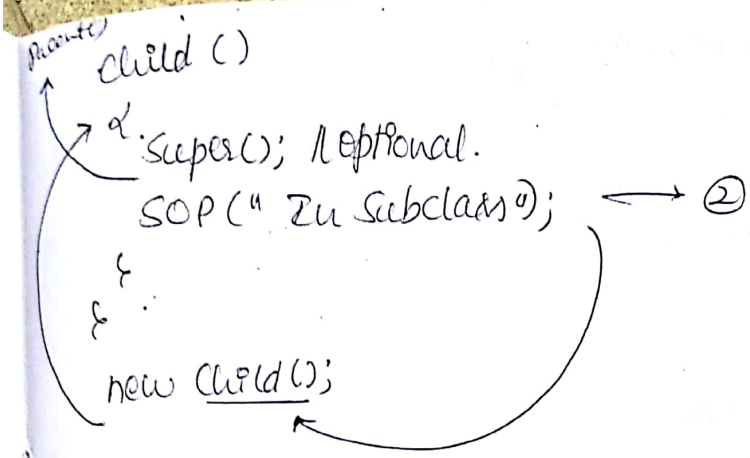
→ Constructor chaining will happen either implicitly or explicitly.

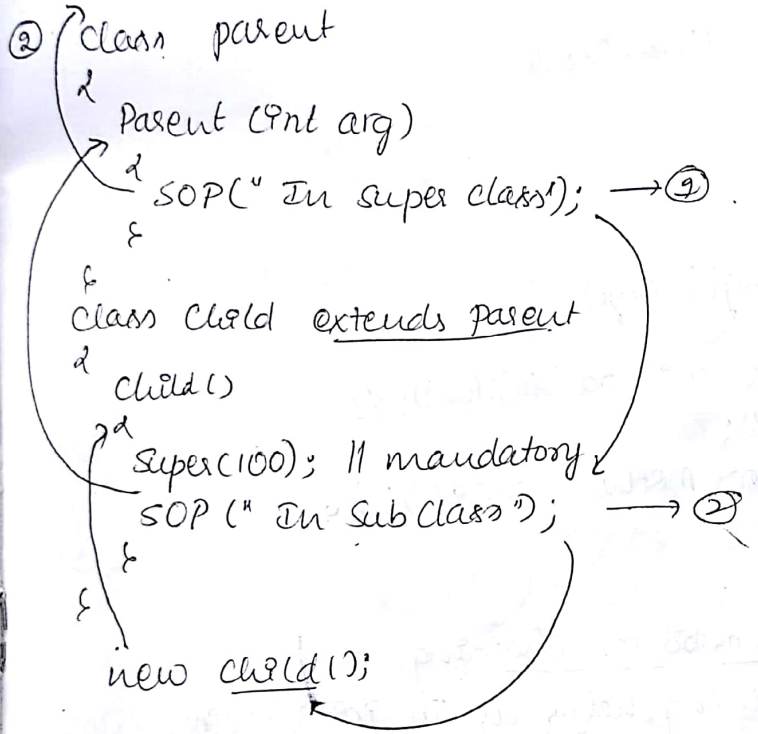
→ If super class has no arguement constructor, implicit constructor chaining will takes place, if the super class has arguement constructor, then user should explicitly take the control to super class constructor by using 'super' with appropriate number and type of arguement

→ Whenever we create an object of a sub class either implicitly or explicitly, super class constructor has to get execute. This is required to initialize super class data member

Eg:

```
Class Parent
{
  Parent()
  {
    SOP("In super class");   → (1)
  }                          → (2)
}
Class child extends parent
{
```





28/09/17

Eg program for Constructor calling & chaining

```
Class A
{
  A(int arg)
  {
    this('A');
    S.O.P("In A class"); //3
  }
  A(char arg)
  {
    S.O.P("In A class char arg"); //2.
  }
}
Class B extends A
{
  B()
  {
    super(212); // mandatory
```

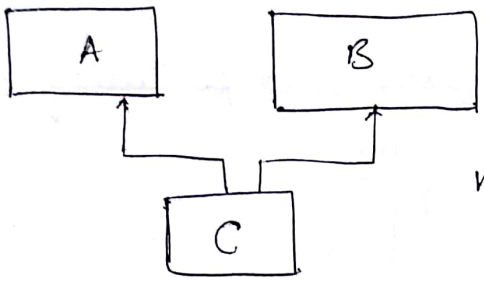
```
    S.O.P("In B class"); //4
  }
}
class C extends B.
  {
    C(double arg)
    {
      super(); // optional
      S.O.P("In C class"); //5
    }
  }
class progrm4
{
  P.S.V.m(String[] args)
  {
    S.O.P("Main method started"); //1
    new C(323.43);
    S.O.P("Main method ended"); //6.
  }
}
```

* Advantages of Constructor chaining

→ Constructor chaining helps us in initializing super class data members.

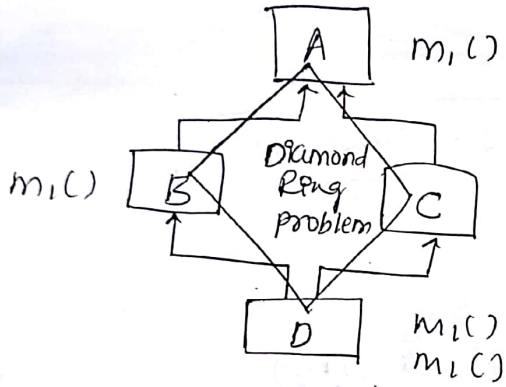
* Why Multiple Inheritance is not possible in Java.?

Since Multiple Inheritance will have more than 1 super class, whenever we create an object of subclass either implicitly or explicitly super class constructor has to be called to complete constructor chaining process. But in multiple inheritance, BC2 we will have more than 1 Super classes, there is an ambiguity in called super class constructor. Hence Multiple Inheritance is not possible in Java.



new C(); // ambiguity in Calling constructor of Super Class.

* Diamond Ring Problem.



D d = new D();
d.m1(); // Ambiguity problem.

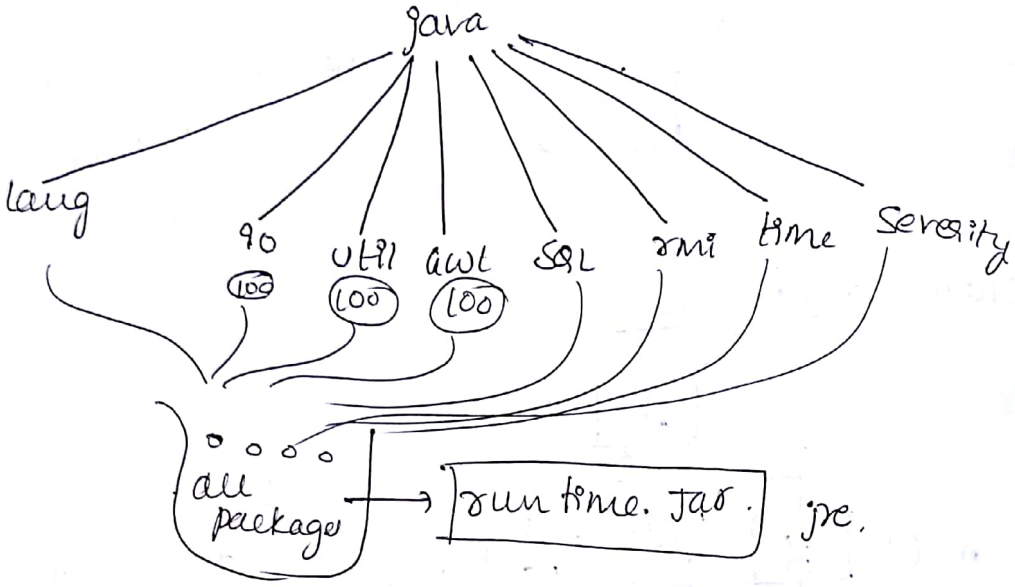
All with same name hence compiler find ambiguity problem.

* Differences

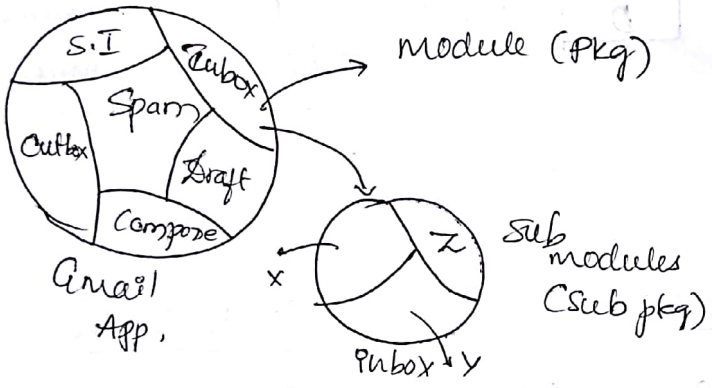
this, Super	this(), Super()
⇒ this → used to refer current object member super → used to refer static class member from S.C	⇒ this() → used to call one constructor from another constructor of same class super() → used to called super class constructor from Sub class.
⇒ These keyword should be used either within non-static method body or inside Constructor	⇒ These keyword should be used within Constructor only as first stmt only
⇒ Can be used together	⇒ Can't be used together.

* Packages

Collection of Classes and Interfaces.



03/10/17



→ In java, package is a collection of classes & Interfaces

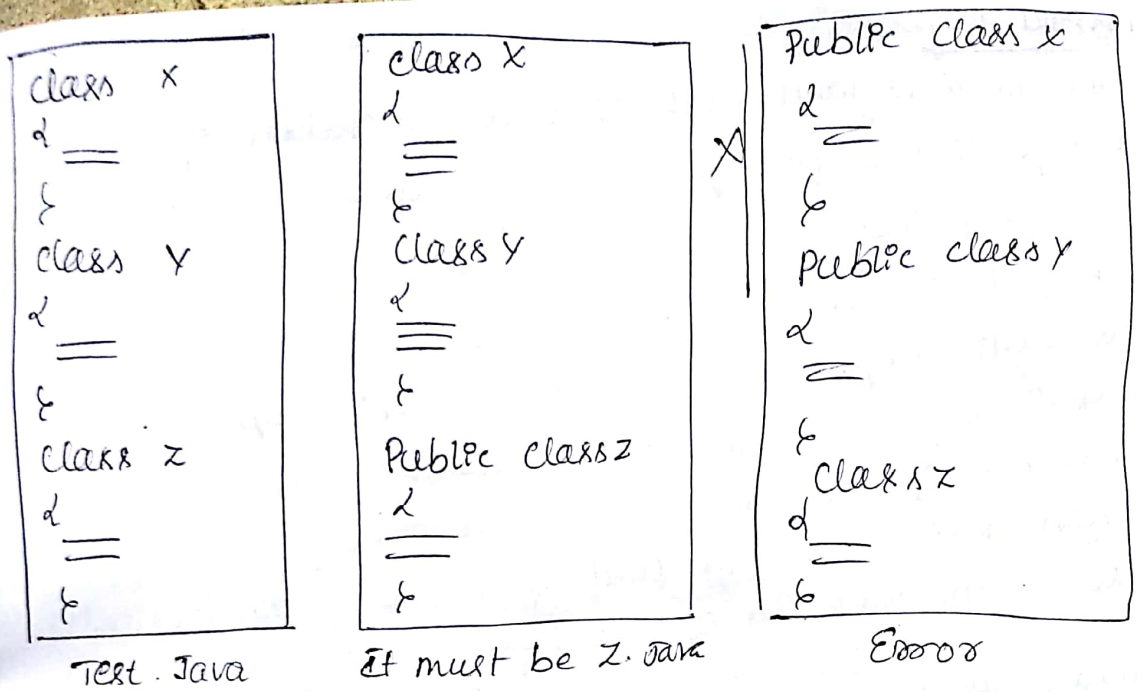
→ A package can be created by using the keyword package.

Eg: Syntax: `package packageName;`

* package pk1;
* package Gmail.login;
  p.pkg C.pkg
* package Gmail.inbox;
* package Gmail.sent;

→ Package creation statement should be the first statement in the source file

→ A package can contain any number of classes & interfaces.



class X ⟶ It should be either public or default.
{
  class Y → Any Access Modifier
  {
  }
}

- Per source file any number of class are allowed but it is not recommended. (usually 1 per source file)
- Per source file, atmost 1 public class is allowed. For such source file we should compulsorily save with the public class name only.
- More than 1 public classes are not allowed in a single source file
- We can specify a class by using any any access modifier provided the class is an inner class. For the outer class we can specify either default or public. (Private & Protected are not allowed).

* IMPORT Keyword.

→ It is a keyword used to access members of one class belongs to one package inside another class belongs to another package.

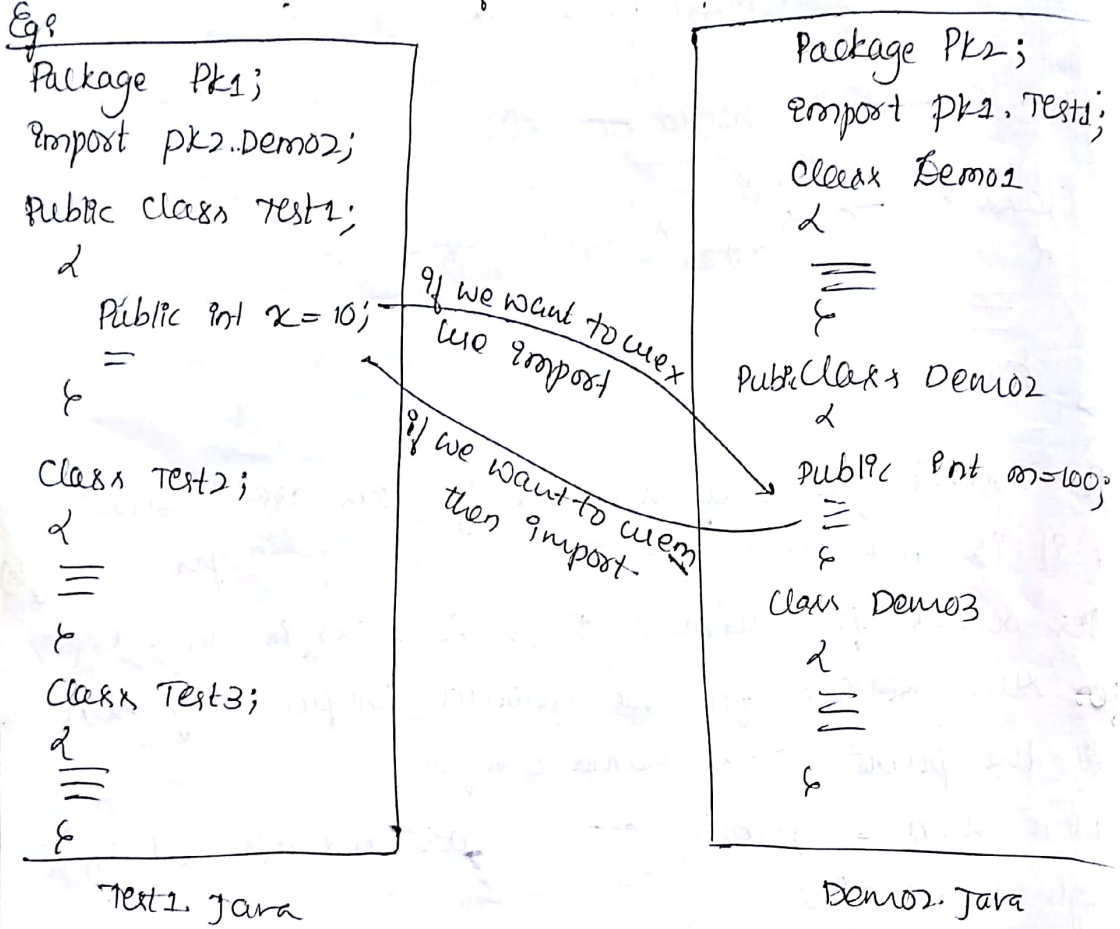
→ Import can be of 2 types :

① Static import ② Non static Import.

→ Static import is used if we want to access static members of the class.

→ Non static import is used if we want to access Non-static members of the class.

Eg:



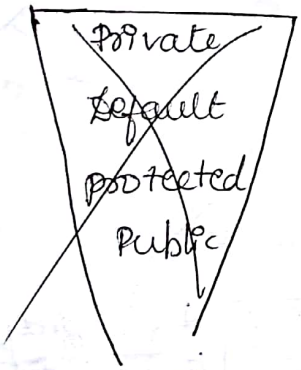
import java.util.Scanner; // Non-static import

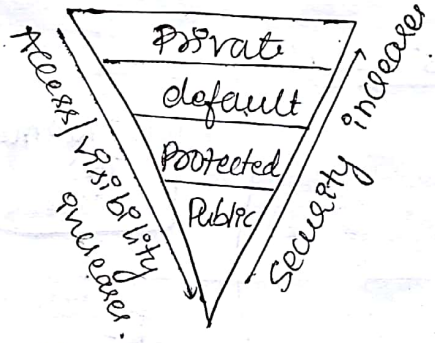
P.pkg C.pkg

import static java.util.Scanner; // Static import.

If I want import all the packages of util package.
import. java. util. *

04/09/17
→ Just because we have imported a class from other package, we'll not be able to access everything from that class. What members we can access depends upon the access ~~speci~~ modifier associated with that member.
→ Java provides 4 different type of access modifier,
(a) Private: These are accessable only within the class.
(b) Default: It is also known as package level access modifier. Default members are accessable within the package from any class.
(c) Protected:- It is same as Default + we can access protected members outside the package provided, the class where we are trying to access should have IS-A relationship with the class to which that members belong to.
(d) Public: These are accessable from any class of any package by doing import.



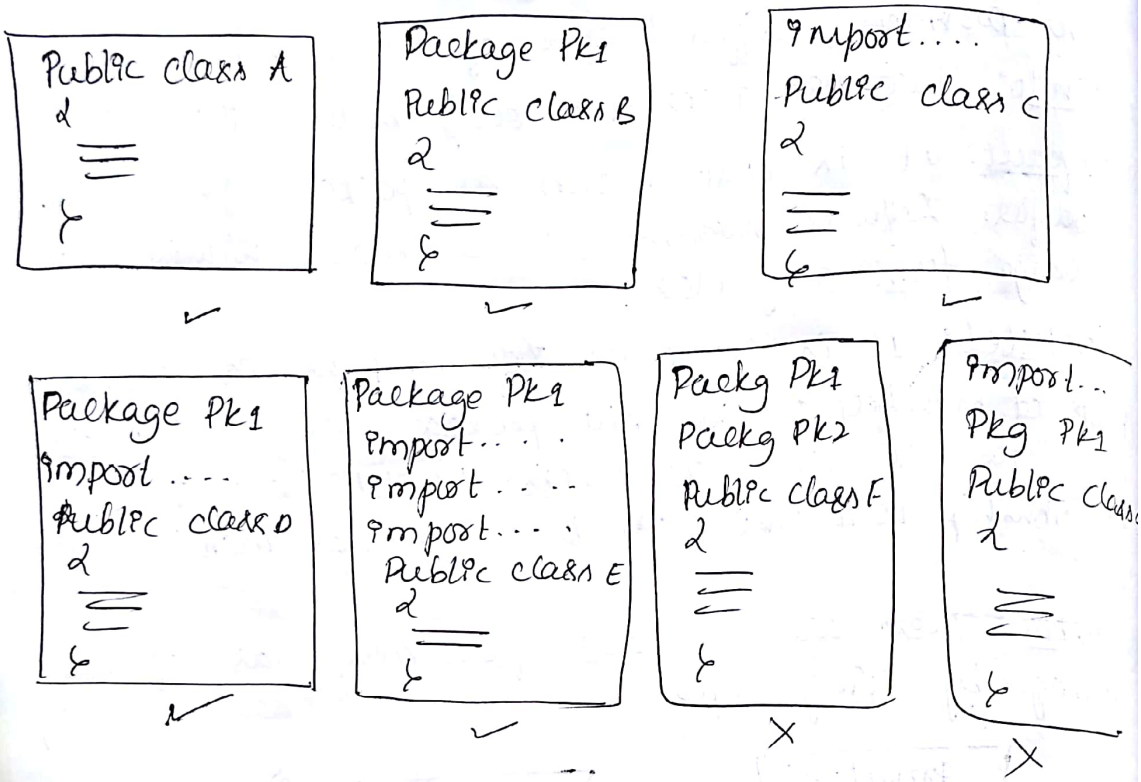


Note:
* If the class is Public, does not mean members are also public.

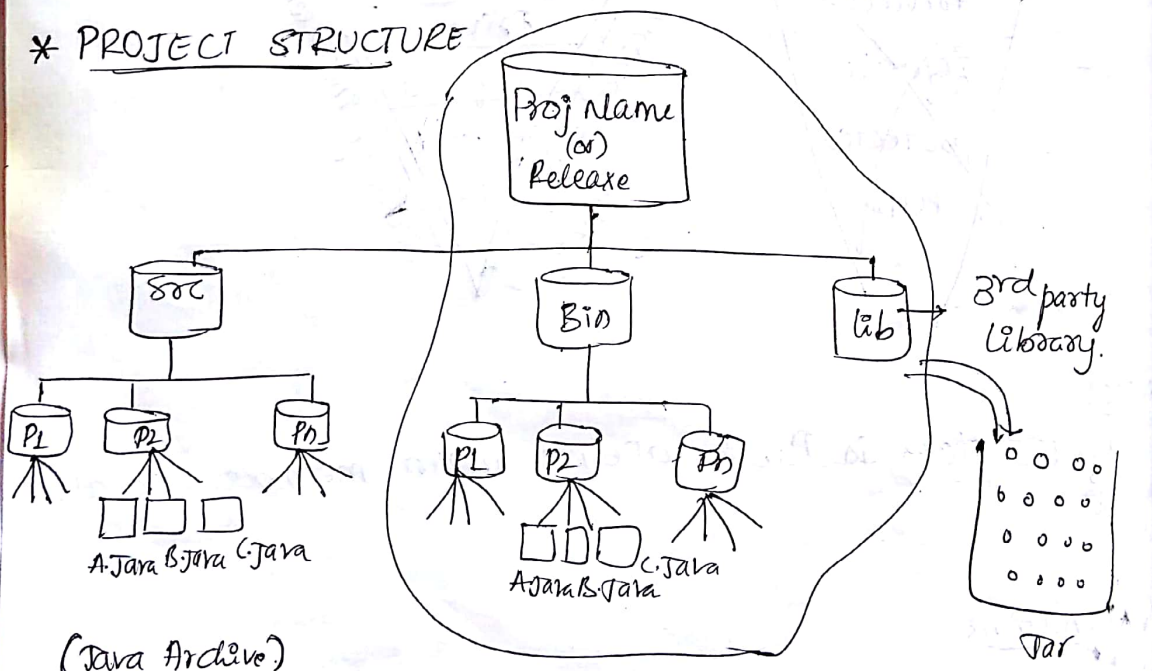
* Definations
→ Data Hidding: Process of hidding data members of the class from outside the class.

Encapsulation: Binding data members of the class with the member functions of the class into a single entity is known as encapsulation.

* Jar file is an encapsulation of bytecode file.
* Package is an encapsulation of class.
* Class is an encapsulation of data members & member functions.



* PROJECT STRUCTURE



(Java Archive)

→ Jar file is given to T.E to Test. Once Jar file is given developer job is done.

05/10/17

Eg: Hierarchy

ProjectName
- Src
  - gmail.login
    - * Mainclass.java
    - * Test01.java
    - * Test02.java
  - gmail.logout
    - * Test03.java

Test01.java

```
Package gmail.login;
Public class Test01
{
    Private int m=10;
            int x=20;
    Protected int y=30;
    Public int z=40;
}
```

Mainclass.java

```
Package gmail.login;
import gmail.logout.Test03;
Public class Mainclass
{
Public S.V.M(String[] args)
{Test02
S.O.P("M.M.S");
Test02.t1=new test02();
t1.print();
Test03 t3=new test03();
t3.view();
S.O.P("Main M E");
}
}
```

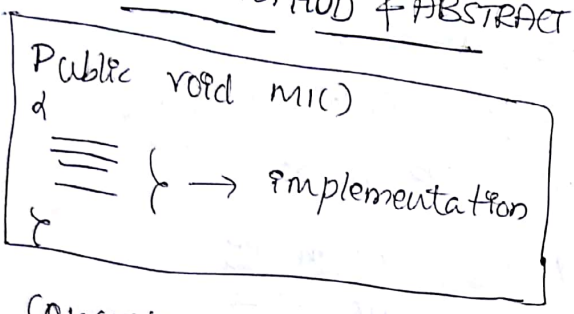
Test02.java

```
Package gmail.login;
Public class Test02
{
  Public void print()
  {
   Test01 t1=new Test01();
   System.out.println("In same Pkg");
   system.out.println("t1.x="+t1.x);
   System.out.println("t1.y="+t1.y);
   System.out.println("t1.z="+t1.z);
  }
}
```

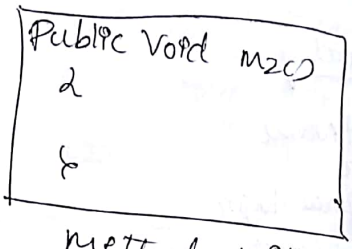
Test03.java

```
package gmail.logout;
import gmail.login.Test01;
Public Class Test03 extends Test01
{
  Public void view()
  {
    S.O.P("In another pckg");
    S.O.P("y="+y);
    S.O.P("z="+z);
  }
}
```

* ABSTRCT METHOD & ABSTRACT CLASS.



Concrete method



method with empty implementation.

method with no implem°.

→ A Method which doesnot have any defination is known as Abstract Method. Such method we should compulsorily declare using the keyword ABSTRACT

→ A class declared by using the keyword abstract is known as Abstract class

→ If a class contain atleast 1 abstract method, compulsorily we should declare the class as Abstract

→ If the class donot contain any abstract method, then it is optional to declare such class as abstract.

→ Abstract class object cannot be created. i.e., we cannot instantiate the abstract class.

Eg:

(optional)
```
abstract class A
{
    P S V m1()
    {
    }
    PS V M2()
    {
    }
}
```

(compulsory)
```
abstract class B
{
    abstract P S V m1();
    abstract P S V m2();
}
```

→ We Provide defination for all the abstract method of abstract class in the subclass. Otherwise declare the subclass as Abstract

Eg: A.java

```
Package jsp.abstract;
abstract Public Class A
{
  abstract Public void m1();
  abstract public void m2();
}
```

Hierarchy

→ Project Name
  → src
    → jsp.abstract
      * A.java
      * B.java
      * Mainclass1.j

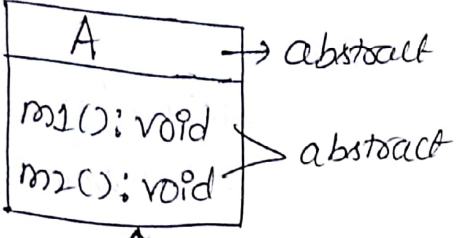
B.java

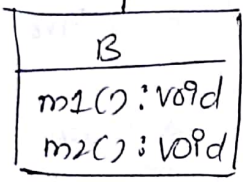
```
Package jsp.abstract
Public class B extends A
{
  public void m1()
  {
    S.O.P ("In m1() method");
  }
  Public void m2()
  {
    S.O.P ("In m2() method");
  }
}
```

Main class.java

```
Package jsp.abstract ——→ { Public class Mainclass1
Public static void main (String[] args)
{
  S.O.P ("Main method started");
  B b = new B();
  b.m1();
  b.m2();
  S.O.P ("Main method ended");
}
}
```

O/P: M M S
In m1() method
In m2() method
M M E

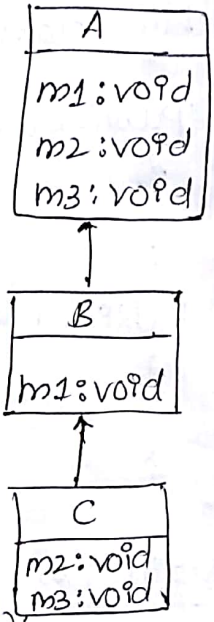




→ Provide def^n for both m1() & m2() otherwise decrease 'B' as abstract.

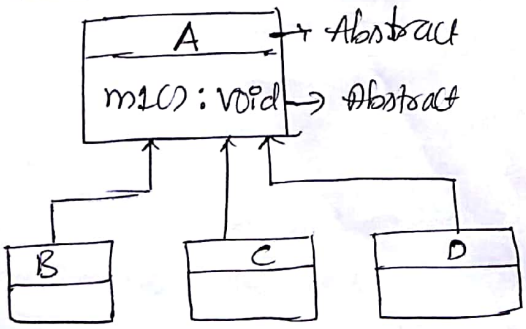
Eg:

```
abstract class A
{
  abstract void m1();
  abstract void m2();
  abstract void m3();
}
abstract class B extends A
{
  Public void m1()
  {
    ==
  }
}
class C extends B
{
  void m2()
  {
    ==
  }
  Protected void m3()
  {
    ==
  }
}
```




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

06/10/17

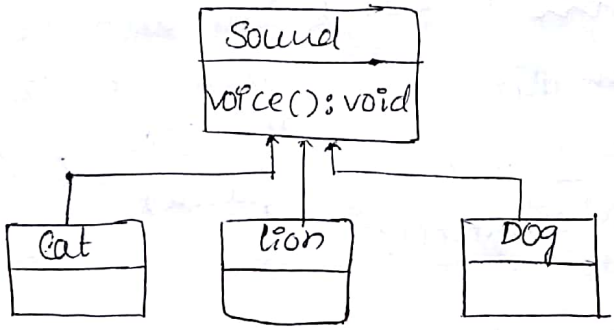


```
class B extends A
{
  public void m1()
  {
    SOP("Hi");
  }
}

class C extends A
{
  public void m1()
  {
    SOP("Bye");
  }
}

class D extends A
{
  public void m1()
  {
    SOP("Hello");
  }
}
```

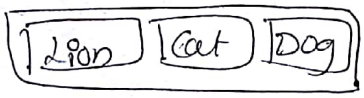
Eg:



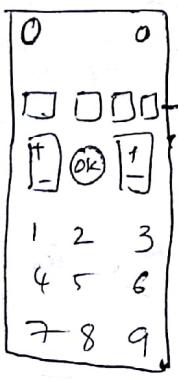

```
class Cat extends Sound
{
  public void voice()
  {
    SOP("Meow Meow");
  }
}

class Lion extends Sound
{
  public void voice()
  {
    SOP("Roar");
  }
}

class Dog extends Sound
{
  public void voice()
  {
    SOP("Bow Bow");
  }
}
```



Eg:



→ Abstract buttons which may get future functionality

Eg:

```
Class DB2
{
  Public void Connect()
  {
    // Code to Connect ~~DB2~~ ~~SQLServer~~ ORACLE
  }
  Public void disConnect()
  {
    // Code to disConnect ~~DB2~~ ~~SQLServer~~ ORACLE
  }
}
```

Using abstract method + Class (Code need not to be removed instead retained and used if required in future).

```
abstract class Connection
{
  abstract void Connect();
  abstract void disconnect();
}

class DB2 extends Connection
{
  Public void Connect()
  {
    // Code to Connect DB2
  }
  Public void disConnect()
  {
    // Code to disConnect DB2
  }
}

class ~~DB2~~ SQLServer extends Connect
{
  Public void Connect()
  {
    // Code to Connect SQL
  }
  Public void disConnect()
  {
    // Code to disConnect SQL
  }
}
```

* Important CONCLUSIONS :-

(1) Abstract class should never be declared as final. Because final class will never have a subclass whereas abstract class should have a subclass.

(2) Abstract keyword should be used only for methods and classes But not for data members.

(3) static methods cannot be declared as Abstract and vice versa. i.e., Abstract and static modifiers will never exist together.

Reason: Abstract method defination should always be provided in subclass. But static methods will never inherit to subclass. and hence static method cannot be declared as abstract class.

(4) 3/10/17

(4) Though the abstract class object cannot be created, abstract class will have a constructor. This constructor is used to initialize the data members of the abstract class through constructor chaining process.

Eg: Test01.java

```
Package jsp.Abstract
abstract public class Test01
{
  int x;
  int y;
  Test01 (int x, int y)
  {
    this.x = x;
    this.y = y;
  }
  abstract public void add();
}
```

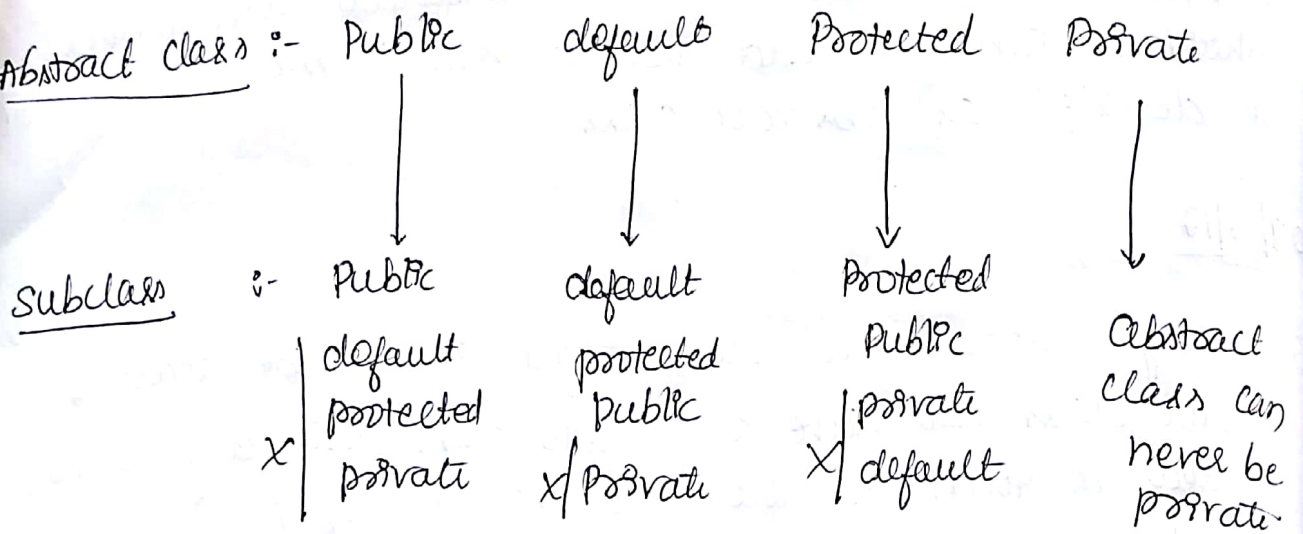
Test02.java

```
package jsp.abstract
public class Test02 extends
                        test01
{
 Test02()
 {
   Super(10,20);
 }
 Public void add()
 {
   S.O.P("Sum="+(x+y))
 }
}
```

Mainclass.java

```
package jsp.abstract.
 public class Mainclass2
 {
    P.S.V.m (String[] args)
    {
       S.O.P ("****");
       Test02 t2 = new test02();
       t2.add();
       S.O.P ("****");
    }
 }
```

(5) While providing the defination for the abstract method of the abstract class either we should retain the same access modifier or increase the visibility to the subclass. (we cannot decrease the visibility)



* Important question on Abstract method & class

① What is abstract method and abstract class?
② Where we provide defination for abstract method?
③ When we should declare method as abstract?
④ Can the abstract method be static. Explain?
⑤ Can a declare abstract class as final. Explain?
⑥ Does abstract class allows constructor. If yes, why?
⑦ Why can't we create an object of abstract class?

There is a chance that abstract class can have only ~~some~~ abstract methods which doesn't contain any defn. Hence there is no point in creating the object of abstract class.

* INTERFACES :-

→ This is a java definition block used to define only abstract methods. (No concrete methods are allowed)

→ Interface data members are by default static & final whereas methods are by default public and abstract

SYNTAX :-

```
Interface InterfaceName
{
// Data members are by default static & final.
// Member functions are by default Public & Abstract
}
```

Eg:

```
Interface A
{
static final int x=10;        (Mandatory / Explicitly declare)
Public abstract void m1();
}
```

⟺

```
Interface B
{
int x=10;
void m1();
}
```

→ We Provide the implementation for all the abstract method of interface in the implementation class by using implements keyword.

→ If implementation class is not providing implementation for all the abstract methods, then declare the implementation class as abstract

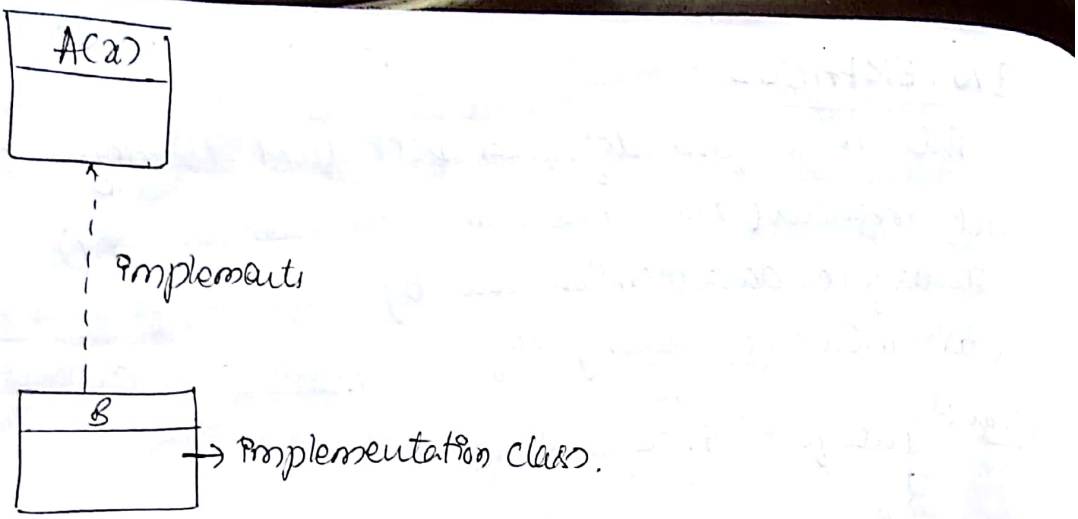
Eg:

```
Interface A
{
int x=10;
void m1();
}
class B implements A
{
Public void m1()
{
==
}
}
```

B b = new B();
b.m1();
S.O.P(A.x); //10

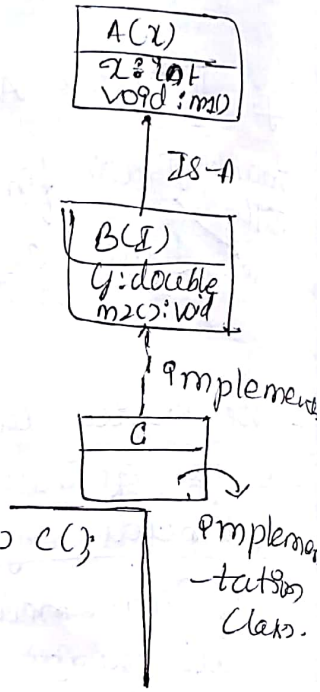
→ mandatory (else error) As per conclusions

→ By default public since interface.



→ An interface can extend from another interface but cannot extend from a class.

Eg:
```
interface A
{
  int x=10;
  void m1();
}
interface B extends A
      (I)        (I)
{
  double y=8.14;
  void m2();
}
class C implements B
    (C)          (I)
{
  Public void m1()
  {
    ==========
  }
  Public void m2()
  {
    ==========
  }
}
```

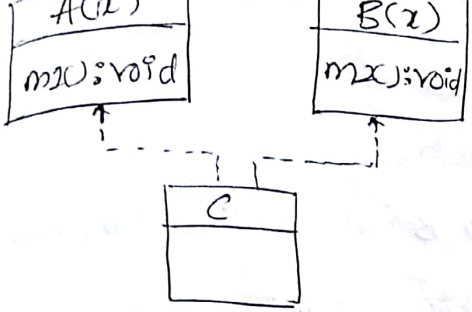



```
C c= new C();
C.m1();
C.m2();
S.O.P(B.x); //10
S.O.P(B.y); //8.14
B.x=100 } X final
B.y=3.41 }
```

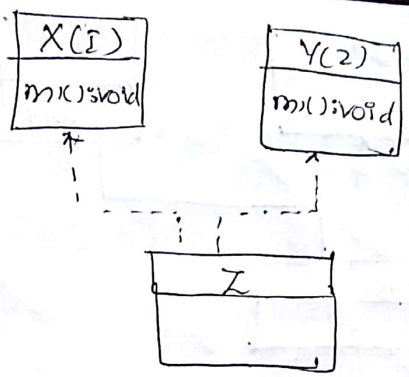
10/10/17

→ A class cannot extend from more than 1 class but it can provide ~~interface~~ implementation for more than 1 interfaces.

Eg:



class C implements A, B
{
provide implem^n for
both m1() & m2()
}



class Z implements X, Y
{
public void m1()
{
==
}
public void m1()
{
==
}
}

X not Possible

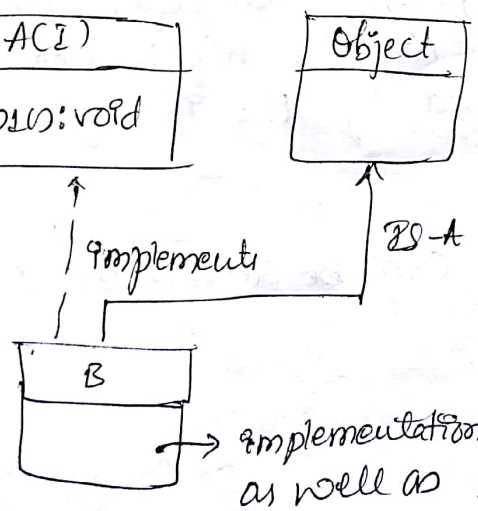
Z z = new Z();
z.m1(); // ambiguity

→ When a class implement more than one interface & if those interfaces are having method with same signature then implementation class should provide implementation for only one method.

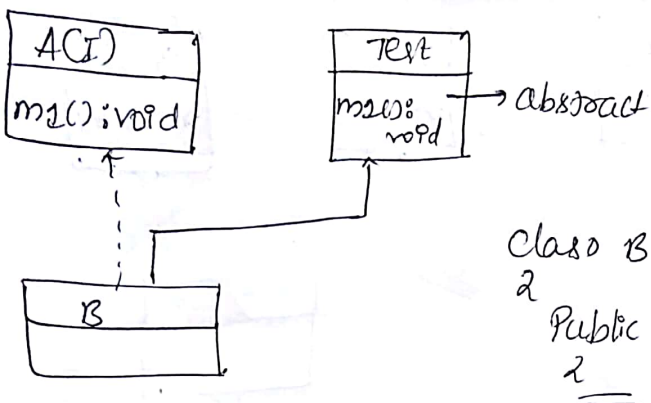
If the interfaces are having methods with different signature, then implementation class should provide implementation for all the methods.

→ A class can extend from another class and implements from an interface simultaneously

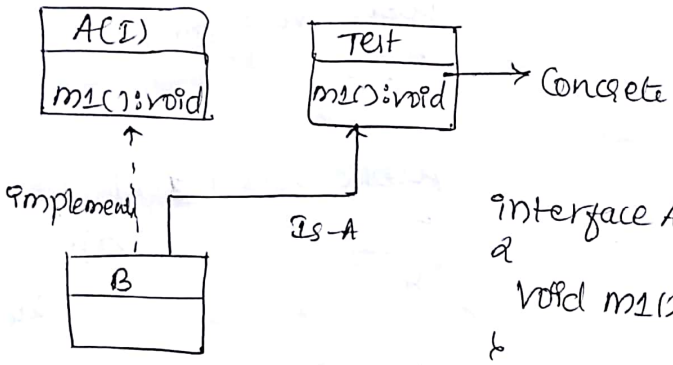
Eg:



class B extend Object
implements A
{
public void m1()
{
==
}
}




```
Class B extends Test implements A
{
  Public void m1()
  {
  }
}
```




```
interface A
{
  void m1();
}
```

```
Class Test
{
  Public void m1()
  {
    SOP("Hi");
  }
}
```

```
Class B extends test implements A
{
// No need to provide defn for m1 of
  interface : Test has Concrete
  method with same interface
}
```

→ An interface can have any number of implementation class. Different implementation classes can have different defn for the same abstract method of an interface.

→ We can use an interface type reference variable to make it to refer to any of its implementation class object. Which implementation class behavior will be executed depends upon the object referred by the interface type reference.

Note: Interface type reference can be created but not an object

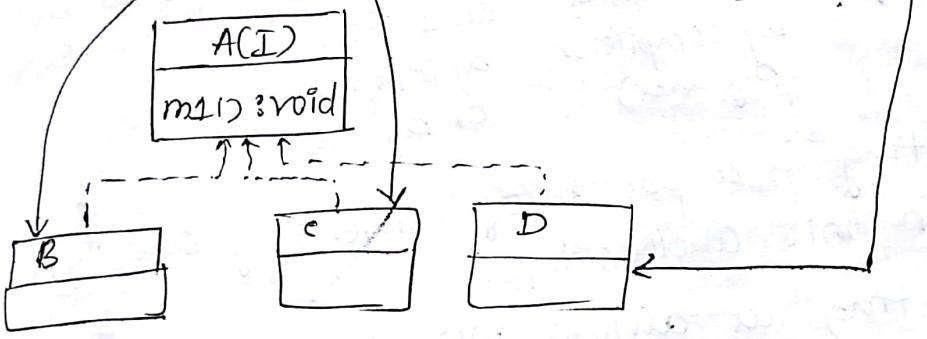
Eg: B b = new B();
b.m1();
C c = new C();
c.m1();
D d = new D();
d.m1();

A a; // a is interface type
new A(); ✗

a = new B(); ✓
a.m1();
a = new C(); ✓
a.m1();
a = new D(); ✓
a.m1();

interface type ref

implementation class.



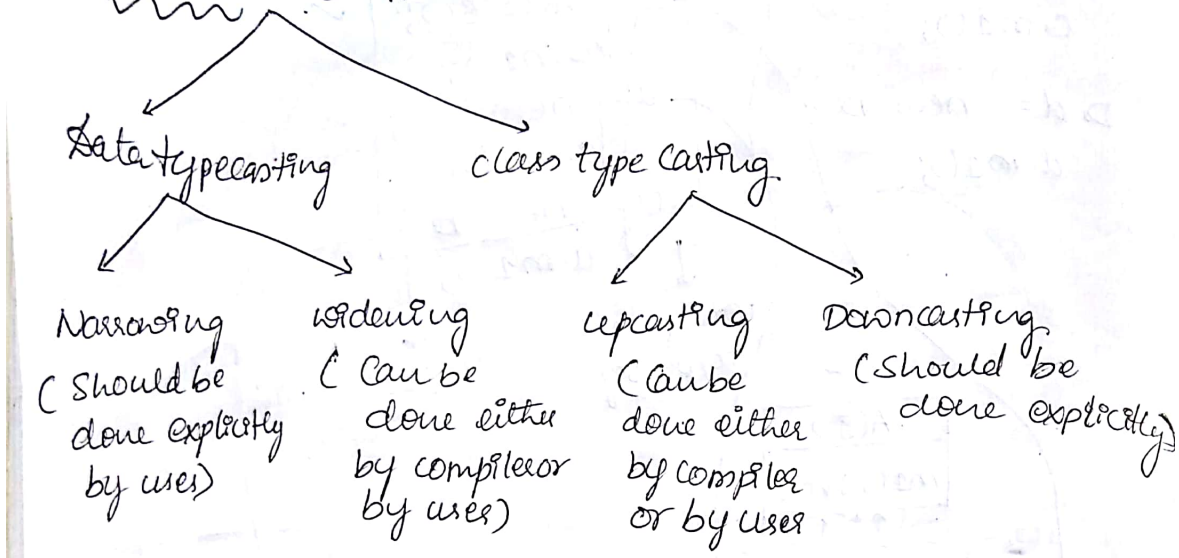
* Differences

Abstract class	Interface
→ Contain both concrete & abstract method	→ Contain only abstract method (till 1.7v)
→ Will have constructor	→ There is no constructor.
→ Data members can be static or Non-static	→ By default static & final
→ Defined in subclass using extends keyword.	→ Definition provided in implementation class using implements keyword.

Type casting.

11/10/17

TYPE CASTING:-



→ Type casting is a process of converting one type of information into another type

→ Typecasting is divided into two types as shown above.

* Data type casting :-

→ It is a process of converting one type of data to another type.

→ Widening :- It is a process of converting smaller type of information to larger type. This can be done either implicitly by the compiler or explicitly by the user. (BCZ it doesnot involve loss of data).

→ Narrowing :- It is a process of converting larger type of data into smaller type. Since it involves loss of data, it should be done explicitly by the user.

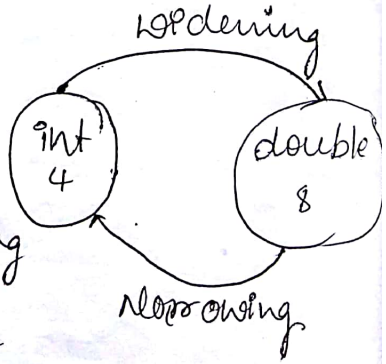
→ Data typecasting is required only for type mismatching statements.

Eg: int x = 10;
double y = 3.14; } Type matching Stmts

int m = 3.14;
double n = 100; } Type mismatching Stmts.

① → Implicit widening.

②



double d = (double) 100; // explicit widening

② int m = (int) 3.14; // explicit Narrowing

float f = 8.14;

float f = (float) 8.14; // explicit Narrowing

* Class typecasting:-

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

→ In order to do the class type casting we need the following two things should exist.

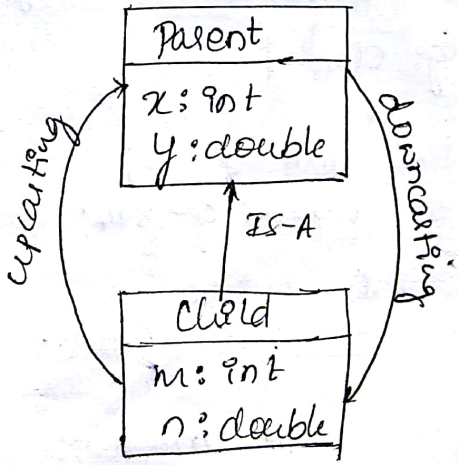
ⓐ We should have IS-A relation-ship with b/w the class.

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

→ Upcasting: It is a process of converting subclass type of information to super class. type. Since subclass contains the properties of the superclass, either compiler or user can do upcasting.

→ Downcasting: It is a process of converting super class type of information to subclass type. Since superclass doesnot contain the properties of subclass, downcasting causes an exception. (Class Cast exception).

Eg:
```
class Parent
{
  int x=10;
  double y=3.14;
}
class Child extends Parent
{
  int m=20;
  double n=8.14;
}
```



Parent P_1 = ~~child~~ new Parent(); } type matching stmts
Child C_1 = new Child();

Parent P_2 = new Child(); } type mismatching stmts
Child C_2 = new Parent();

auto/implicit upcasting

$P_2.x$ } ✓
$P_2.y$ }

$P_2.m$ } X
$P_2.n$ }

P_2 → x=10, y=3.14, m=20, n=8.14 (Hidden)

Child C_2 = (Child) new parent();
↓
class cast exp.

C_2 → x=10, y=3.14

→ When a super class reference is pointed to a subclass object, though object contains properties of superclass and subclass, using superclass reference we can access only the properties of superclass (subclass properties will be hidden)

→ To access the subclass properties, we need to perform downcasting on an upcasted object. i.e., downcasting should be done only after doing upcasting

Eg:
```
class Parent
{
  int x=10;
  double y=8.14;
}
class Child extends Parent
{
  int m=20;
  double n=3.14;
}
```

```
Parent P2 = new Child();   //upcast
P2.x }
P2.y } ✓

Child C2 = (Child) P2;   //downcast
C2.x;
C2.y;
C2.m;
C2.n;
```

P_2 → x=10, y=3.14, m=20, n=8.14 object

Parent P_1 = new Parent(); // Type matching.

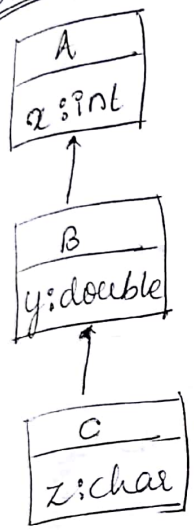
Child C_1 = (Child) P_1; // Class Cast exp

Child C_2 = (Child) new Parent(); // CCE (same as above 2 stmt)

Parent P_2 = new Child(); // upcasting

Child C_3 = (Child) P_2; // down casting (No exp)

13/10/17



Case 1:
C C_1 = new C(); //
C C_2 = (C) new B(); } CCE
C C_3 = (C) new A();

Case 2:
B b_1 = new B(); //
B b_2 = new C(); // upcasting
B b_3 = (B) new A(); // CCE

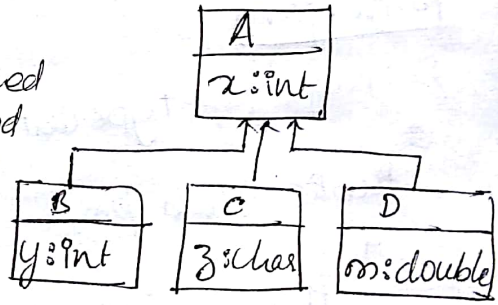
Case 3:
A a_1 = new A(); //
A a_2 = new B(); //
A a_3 = new C(); //

Type matching

upcasting.

→ Advantage of Type casting.

```
class point
{
  public void disp(A a) // Generalized method
  {
    a.x;
    if (a instanceof B)
    {
      B b = (B) a;
      b.y;
    }
    else if (a instanceof C)
    {
      C c = (C) a;
      c.z;
    }
    else if (a instanceof D)
    {
      D d = (D) a;
      d.m;
    }
  }
```



point P = new point();

P.disp(new A()); ⇒ A a = new A();
P.disp(new B()); ⇒ A a = new B();
P.disp(new C()); ⇒ A a = new C();
P.disp(new D()); ⇒ A a = new D();
P.disp(new object()); ⇒ A a = new object();

↓ error

→ Instanceof: * It is a keyword used to check whether the object contains the properties of the specified object.
* It returns true if object contains the values of specified object or else false.

eg: A.java

```
package jsp.typecasting;
class A
{
  int x=10;
}
```

B.java

```
package jsp.typecasting;
class B
{
  int y=20;
}
```

C.java

```
package jsp.typecasting;
class C
{
  int z=30;
}
```

D.java

```
package jsp.typecasting;
class D
{
  int m=40;
}
```

Print.java

```
Package jsp.typecasting;
Public class Print
{
  Public void disp(A a) // Generalized method
  {
    S.O.P("A.x=" +a.x);
    if(a instanceof B)
    {
      B b = (B) a;
      S.O.P("B.y = "+b.y);
    }
    if (a instance of C)
    {
      C c =(C) a;
```

```
if ( a instanceof D)
{
  D. d = (D) a;
  SOP(" D.m = " +m);
}
```

Main class. java

```
Package jsp.typecasting
public class MainClass
{
  P S V m (String[] args)
  {
    SOP(" m m s")
```

→ Advantages of type casting

⇒ It helps us in achieving Generalization

⇒ It helps us in writing generalized cache block

⇒ It helps in achieving dynamic polymorphism through method overriding.

14/10/17

* Private Constructor.

→ Constructors by default will have the same access modifier as that of the class. However we can provide any access modifier to a Constructor.

→ If we declare constructor as private it will not allow us to create an object outside the class.

Eg: Package jsp.singleton;

```
Public class Test01
{
  Private Test01()
  {
    S.O.P (" Running Constructor ");
  }
  Public void disp()
  {
    S.O.P (" Running disp() method ");
  }
```

```
Public static Testo1 getobject()
{
    return new Testo1();
}
```

Main Class

```
Package jsp.singleton;
Public class MainClass1
{
    Public static void main(String[] args)
    {
        S.O.P("M.M S");
        Testo1 t1 = testo1.getobject();
        Testo1 t2 = testo1.getobject();
        SO.P("MM E");
    }
}
```

* Singleton ~~Constructor~~ Class

A class which allows us to create a single object throughout the lifetime of the application is known as singleton class.

In order to create a singleton class first we need to declare the constructor as private then write the necessary logic which prevents the class to create multiple object. ~~Then~~ ~~write~~ the

eg:

```
package jsp.Singleton  → Public class SingletonClass
{
Public static SingletonClass s;
private int count;

Private SingletonClass()
{
  SO.P("Running Constructor");
   Count++;
}

Public static SingletonClass getInstance()
{
```

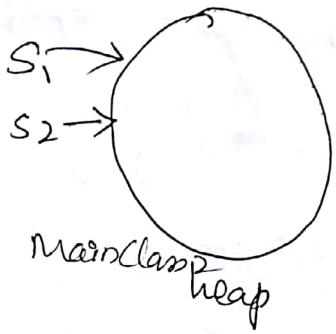
if (

{

s = new singletonClass();

}

}



Mainclass

```
package jsp.Singleton
public class MainClass
{
psvm(String[] args)
{
S.O.P("M M S");

SingletonClass S1 = SingletonClass.getInstance();
SingletonClass S2 = SingletonClass.getInstance();
S.O.P("M.M.E");
}
}
```

O/P: M M S
Running Constructor
M M E

* Java Bean Class:

→ writing a class with public access modifier, private data members, Public Constructor and getter's and setter's in known as Java Bean class.

→ Getter's are used to get the information. Whereas setter's are used to set the information for the attributes (properties).

→ Java Bean Class is very good class for Data hiding and encapsulation.

→ Java Bean class is used to achieve the DAO (Data Access Object) and DTO (Data Transfer Object) of the design application.

Eg:
```
package jsp.singleton;
public class Student
{
private String name;
private int id;
```

```
this.name = name;
this.id   = id;
this.dept = dept;
```

```
public String getName()
{
  return name;
}
public String getId()
{
  return id;
}
public String getDept()
{
  return dept;
}
```

```
public void setName(String name)
{
  this.name = name;
}
Public void setId(int id)
{
  this.id = id;
}
Public void setDept(String dept)
{
  this.dept = dept;
```

Mainclass

```
package jsp.singleton;  → Public class MainClass3
P.S.V.M (String[] args)
{
  S.O.P ("M M S");

  Student S1 = new Student ("Dinga", 10, "CSE");
  Student S2 = new Student ("Dingi", 20, "ISE");

  S.O.P("Student details !!!");
  S.O.P("S1.id = " +S1.getId());
  S.O.P("S2.id = " +S2.getId());
  S.O.P("S1.dept = " +S1.getDept());
  S.O.P("S2.dept = " +S2.getDept());
  S.O.P("S1.name = " +S1.getName());
  S.O.P("S2.name = " +S2.getName());

  S2.setDept("CSE");
  S.O.P("S1.name = " +S1.getName());
  S.O.P("S2.name = " +S2.getName());
}
```

O/P:

Data member

Static
- Static Block
- Instance Block
- Constructor
- non Static method
- Static method

Non-Static
- Instance Block
- Constructor
- Non-Static method.

* Method Overriding:-
+ Whenever we inherit a class from superclass, all the properties of superclass will be inherited to subclass. If we are not happy with any of the implementation of superclass methods, we can always change the implementation in the subclass. This phenomenon of changing the implementation of a super class method in the subclass is known as Method Overriding.
+ While overriding a method in the subclass we should retain the same method signature.

→ Eg:①

```
class Parent
{
    void Marry()
    {
        S.O.P("Parvathi");
    }
}
class Child extends Parent
{
    void marry()
    {
        S.O.P("Mary");
    }
}
```

overrided method

overriding method

Overriding

② Class A
{
void vehicle()
{
S.O.P("Zenna");
}
}

Class B extends A
{
void vehicle()
{
}
}
overriding

Eg③ Class parent
{
void marry()
{
S.O.P("Parvathi");
}
}

Class child extends parent
{
void marry()
{
S.O.P("parvathi");
}
}

It is overriding but called as Stupidity

Eg④ Class parent
{
void marry()
{
S.O.P("parvathi");
}

void ~~no~~dowry()
{
S.O.P("Audi");
}
}

Not overloaded

Class Child extends parent
{
void marry()
{
S.O.P("parvathi");
}
}

overriding

marry

Parent.java

```
package jsp.overriding;
public class Parent
{
  public void marry()
  {
    S.O.P("Parvathi");
  }
  Public void dowry()
  {S.O.P("Audi");
  }
}
```

Child.java

```
Package jsp.overriding
Public child extends Parent
{
  Public void marry()
  {
    S.O.P("Rishika");
  }
  Public void dowry()
  {
    S.O.P("jaguar");
  }
}
```

Mainclass.java

```
Package jsp.overriding
public class Mainclass
{
  P.S.V.M(String[] args)
  {
    S.O.P("M.M-S");
    S.O.P("super class ref & super class obj");
    Parent p1 = new Parent();
    p1.marry();
    p1.dowry();
    S.O.P("super class ref & sub class obj");
    Parent p2 = new child();
    p2.marry();
    p2.dowry();
    S.O.P("M.M.E");
  }
}
```

O/P: Main method started
super class ref & super class object
Parvathi
Audi
Rishika
jaguar

Note:

① Private and static methods cannot be inherited to the subclass and hence they cannot be overrided.

Since main is a static hence we cannot override

② If we declare a method as final, such methods will be inherited to subclass. But subclass cannot change it's implementation. i.e., final methods can be inherited but cannot be overrided.

③

final variable	⟶	can't reassign
final class	⟶	can't inherit from
final methods	⟶	can't overrided

* Difference b/w overloading and overriding

Overloading	Overriding
→ If we want to perform single task in multiple ways then we'll go for overloading	→ If we want to change the task itself, then we go for overriding
→ process of writing multiple methods with the same name but different in either number of arguments or type of argument.	→ It is the process of changing the implementation of super class in sub class
→ Inheritance is not mandatory	→ Inheritance is compulsory
→ Final & static methods can be overloaded.	→ Final and static methods cannot be overrided.
→ Example for Compile time polymorphism or static polymorphism.	→ Example for run time polymorphism or dynamic polymorphism.

Eg: Overriding

* Updation for android
* gmail login page
* Compose page

* Polymorphism

An object showing different states in its different stages of life is known as polymorphism.

polymorphism can be classified into 2 types.

(a) Compile time polymorphism

Also knowing as Static polymorphism. In this type of polymorphism, binding of method declaration with the method defination done by the compiler during compilation time and hence the name compile time polymorphism

Eg: overloading (Constructor + method)

(b) Run Time polymorphism:

Also known as dynamic polymorphism. In this type of polymorphism, binding of method declaration with the method defination is done by the JVM during run-time, based on the run-time object + Hence the name Run-time polymorphism

Eg: method overriding

Section III

* Object class and its methods
→ It is available in java.lang package
→ It is the supermost class in entire java hierarchy
→ It has only no argument constructor.

* Object class methods
→ public String toString()
→ public int hashCode()
→ public boolean equals(Object obj)
→ public final void wait()
→ public final void wait(long ms) (Milliseconds)
→ public final void wait(long ms, int ns)
→ public final void notify()
→ public final void notifyAll()
→ public class getClass()
→ protected Object clone()
→ protected void finalize()
→ private void registerNatives()

Thread related.

* Protected Object clone -

1. If we want to take copy of the object we use all clone method.
2. The Object class clone method does shallow clo
3. An object is eligible for cloning if that obj implements Cloneable interface.
4. Shallow cloning is recommended if there is no Has-A relationship in the classes
5. If the classes are having Has-A relationship it is recommended to perform de-cloning

* Shallow cloning

Eg: Student Java.

```
package jsp.objectclass;
public class Student implements cloneable
{
 String name;
 int id;
 String dept;
 Address a; // HAS-A                                    Address a;
 public Student (String name, int id, String dept,)
 {
  this.name = name;
  this.id = id;
  this.dept = dept;
  this.a = a;
 }
public object clone() throws clone not supported Exception
 {
  return super.clone();
 }
}
```

Address.java

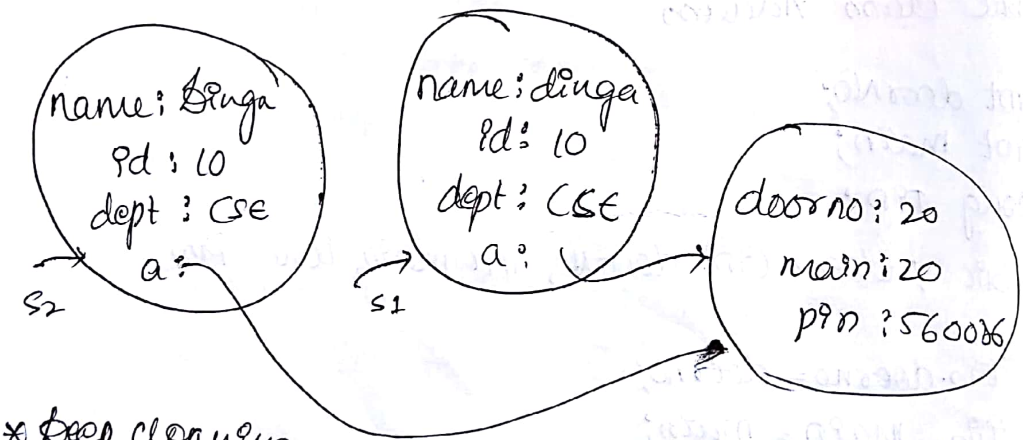
```
Package jsp.object.class;
public class Address
{
 int doorNo;
 int main;
 long pin;
public Address (int doorNo, int main, long Pin)
 {
  this.doorno = doorno;
  this.main = main;
  this.pin = pin;
 }
}
```

S.O.P("*****.....**");

```
Address a= new Address( 20, 20, 560036);
Student S1= new Student( "Dingo", 10, "CSE", a);
S.O.P ("S1 Student details !!");
S.O.P ("- - - - - -------- .");
S.O.P ("Name= "+S1. name);
S.O.P(" Id= "+S1. id);
S.O.P(" dept= "+S1. dept);
S.O.P(" doorno= "+S1. a. doorno);
S.O.P(" main = "+S1. a. main);
S.O.P(" pincode = "+S1. a. pin);
Object obj= S1. Clone ();
Student S2 = (Student) obj; // down casting
S.O.P (" S2 Student details");
S.O.P(" - ------ - - - ");
S.O.P(" Name= "+S2. name);
// changing door no of S2 and pin.
   S1. a. doorno= 30;
   S1. a. pin = 560024;
S.O.P(" after changing S1's address");
S.O.P(" ------ ------- - - - ");
S.O.P (" S2 Student details !!");
```

(but S2.)

(S2.)



* **Deep cloning.**

```
public Object clone() throws CloneNotSupportedException
{
  Address a;
  a= new Address (this.a. doorno, this.a. main,
             this. a. pin);
  return new Student( this. name, this. id, this. dept, a
```

* toString()

It represents ~~str~~ returns string representation of an object. String representation includes fully qualified name of the class. along with hexadecimal equivalent of the # code in the following format

Package Name.ClassName@Hexadecimal equivalent of hashcode

fully qualified

* Public int hashCode():

→ # code is unique integer number associated with an object. This method generates the hashcode based on the hexadecimal address of an object.

→ If 2 objects are having same address, then the hashcode will be same. otherwise different

→ ~~Public Boolean equals(Object obj)~~

* Public Boolean equals(Object obj)

This method is used to compare the current object with the given object based on the hashcode If 2 objects are having same hashcode, then it returns true otherwise false

→ usually we override 2 string method in the subclass to display the states of an object (contents of an object.

→ Usually we override hashcode method in the subclass to generate the hashcode based on unique attribute.

→ Usually we override equals method in the subclass to compare the current object with the given object based on the contents or the states.

Demo1.java

```
Package jsp.objectClass;
Public Class objectDemo1
{
   PSVM (String[] args)
   {
      S.O.P("*********");

      Object obj1 = new Object();
      Object obj2 = new Object();

      String s1 = obj1.toString();
      String s2 = obj2.toString();
      int hc1 = obj1.hashCode();
      int hc2 = obj2.hashCode();

      boolean res = obj1.equals(obj2);

      S.O.P("s1=" +s1);
      S.O.P("s2=" +s2);

      S.O.P("hc1=" +hc1);
      S.O.P("hc2=" +hc2);

      S.O.P("res=" +res);
      S.O.P("*********");
   }
}
```

O/P:

```
*******
s1 = java.lang.object@ hex value of hc1
s2 = java.lang.object@ hex value. hc2
hc1 = 123456
hc2 = 12579.1
res = false.
*******
```

Eg:

```
Package jsp.objectClass;
Public class Mobile
{
   String Name;
   double Price;
   int ram;
}
Public Mobile(String name, double price, int ram)
{
   this.name= name;
   this.price= price;
   this.ram= ram;
}
@override
Public String toString()
{
  // return "Name=" +this.name+ " Price=" +this.price+ " Ram"
   return "Hello stupid";
}
@override
public int hashcode()
{
   return 1000;
}
@override
public boolean equals (Object obj)
{
  Mobile m = (mobile) obj ; // DOWN Casting

   return (this.name == m.name &&
           this.price == m.price &&
           this.ram == m.ram);
}
```

objectDemo2.java.

```
Package jsp.objectClass;
Public Class ObjectDemo2
{
   P S V m ( )
   {
     S.O.P (" ******");
     Mobile m1 = new mobile("Samsung", 30000.00, 4);
     Mobile m2 = new mobile("sony", 30000.00, 4);
     S.O.P (m1.toString());
     S.O.P (m2.toString());
     S.O.P (m1.hashCode());
     S.O.P (m2.hashCode());
     S.O.P (m1.equals(m2));
     S.O.P ("*******");
   }
}
```

O/P:

17/10/17

* write a program to check whether the time in both the watches are same or not. The program should be able to display the time whenever we call 2 string method

Soln

```
Package jsp.objectClass;
Public class Watch
{
   int hh;
   int mm;
   int ss;
  Public Watch (int hh, int mm, int ss)
  {
    this.hh = hh;
   this.mm = mm;
```

```
@Override
public String toString()
{
return this.hh+":"+this.mm+":"+this.ss;
}
@Override
public boolean equals(Object obj)
{
Watch w=(Watch)obj; // downcasting
return this.hh = w.hh &&
this.mm = w.mm &&
this.ss = w.ss ;
```

Main

```
Public Class ObjectDemo;
{
P S V M(String[] args)
{
S.O.P("*********");
Watch w1= new Watch(05,23,32);
Watch w2= new Watch(05,23,43);
S.O.P(w1); // w1.toString()
S.O.P(w2); // w2.toString()
S.O.P(w1.equals(w2));
S.O.P("*******");
}
}
```

* Write a program to check 2 cars are equal or not w.r.t color, model & price. The program should also display the states of the car.

* String Class.

→ It is available in java.lang package
→ It is a final class.
→ String immutable.
→ String class implements comparable interface and hence array of strings objects can be sorted.
→ String is the only class in the entire java whose object can be created both using new and without using new.

Eg: String S = new String();
String S = "jsp";

→ A String class defines overloaded constructor
(i) String S = new String();
(ii) String S = new String("jsp");
(iii) String S = new String(ch);
↳ array of chars

String S = "Hello";
Char[] ch = S.toCharArray();
ch { H | E | L | L | O }

*
→ The following 3 methods of object class have been overrided in the string class.

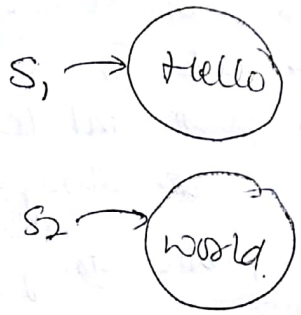
(i) public String toString():- This method has been overrided to display the state of an object (Content).

(ii) Public int hashCode():- This method has been overrided to generate the hashcode based on the state of an object (Content of object). If 2 string objects are having state, hashCode will be same otherwise different

(iii) public Boolean equals(Object obj):- This method has been overrided in the string class, to compare the current object with given object based on state or content of an object

Eg:

```
Public Class StringDemo
{
    p s v m (String[] args)
    {
        S.O.P ("*******");
        String s1 = "Hello";
        String s2 = "World";
        S.O.P("s1 = "+s1); // toString()
        S.O.P("s2 = "+s2);
        S.O.P( s1.hashcode());
        S.O.P( s2.hashcode());
        S.O.P(s1.equals(s2));
        S.O.P ("*******");
    }
```



* Difference b/w equals() and ==

equals()	==
→ Meant for content comparison	→ Meant for reference comparison.

method	Object Class	String Class
toString()	→ To generate string representation of object in fully qualified@hexadecimal	→ to display content/ state of an object
hashCode()	→ based on hexadecimal	→ based on state/

* String class method 23/10/17

→ Public int length()
→ public char charAt(int pos)
→ public char[] toCharArray()
→ Public int indexOf(char ch)
→ public int lastIndexOf(Char ch)
→ Public String substring(int sp)
→ public string substring(int sp, int ep)
→ public boolean Contains(String str)
→ public string toLowerCase()
→ public string toUpperCase()
→ public string[] split(String delimeter)
→ Public String Concat(string str)
→ public int CompareTo(string str)
→ public string replace(String str, char ch)

Eg: Package java.developers

```
Public class StringDemo3
{
  P S V m ( )
  {
    S.O.P ("*********");
    String s = "Java developers";
    S.O.P("s.length() = " + s.length()); // 15
    S.O.P("s.charAt(7) = " + s.charAt(7)); // v
    S.O.P("s.startsWith(Ja)=" + s.startsWith(Ja)); // true
    S.O.P("s.endsWith(rr) = " + s.endsWith(rr)); // false
    S.O.P("s.contains(Dev) = " + s.contains(Dev)); // true
    S.O.P("s.substring(9) = " + s.substring(9)); // lopers
    S.O.P("s.substring(9,14) = " + s.substring(9,14)); // lopers
    S.O.P("s.indexOf('e') = " + s.indexOf('e')); // 6
    S.O.P("s.lastIndexOf('e') = " + s.lastIndexOf('e')); // 12
    S.O.P("*********");
  }
}
```

write a program to count the number of words in a given string and also number of characters in each word.

```
Soln: Public Class StringDemo4
{
    P S V M ( )
    {
      S.O.P("***....");
      S.O.P"
      String S = "Hi Hello How are you";
      String [] str = S.split(" ");
      S.O.P(" total words= "+str.length);
      for(int i=0; i<str.length ; i++)
      S.O.P(i+ str[i]+ " contains "+ str[i].length + " Chars");
      S.O.P("***....***");
    }
}
```

S.split(" ") Convert string to array

Hi	Hello	How	are	you

→ Public int compareTo(String str)

Used to compare current string with given string. It does the comparison by converting the strings into ASCII equivalents. It returns positive number if current string is greater than given string, it returns -ve number if current string is less than given string, it returns 0 if current string = given string

* I

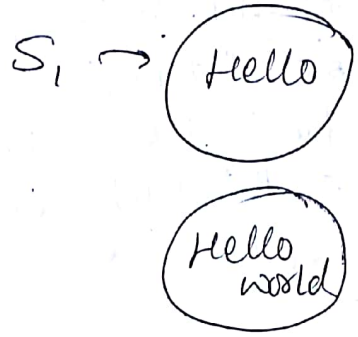
* Immutability.

→ Whenever we create an object of a class we cannot change the state of an object. If we try to change the state, with thoughs new change, a new object will be created. This unchangable behavar is known as immutability

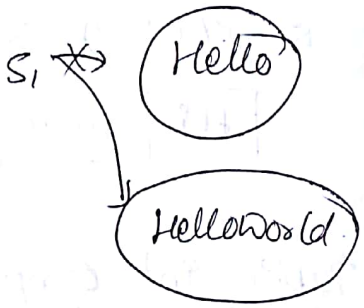
→ String is immutable. i.e., Whenever we try to perform any changes on string object, with thoughs new change a new object will be created.

Eg: ① String s1 = "Hello";
s1.concat("World");
S.O.P (s1);

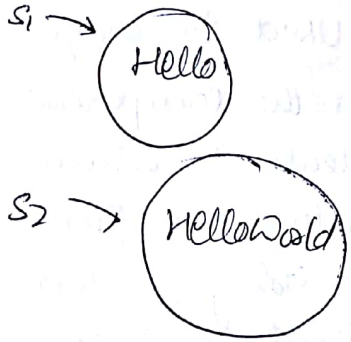
O/P: Hello



② String s1 = "Hello";
s1 = s1.Concat ("World");
SOP(s1); // HelloWorld

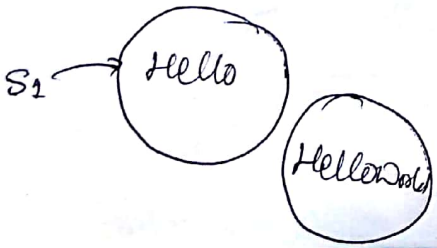


③ String s1 = "Hello";
String s2 = s1.Concat ("World");
S.O.P(s1); // Hello
S.O.P(s2); // HelloWorld



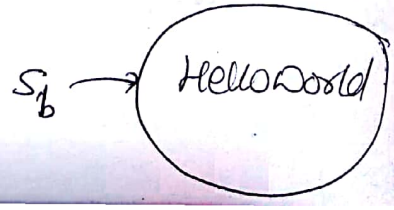
Append.

String s1 = new String ("Hello");
s1.concat("World");
SOP(s1); // Hello



Mutable

StringBuffer sb =
new StringBuffer ("Hello
sb.append("World");
SOP(sb); // HelloWorld

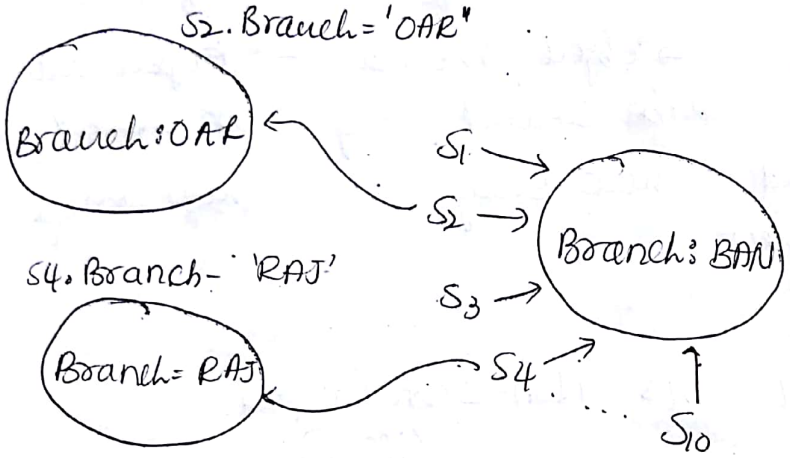


24/10/17

* Why string is immutable?

Whenever a string object is referred by multiple references, if we change the state by using one references it should not affect others. This is possible only if the object is immutable object

Eg:



Since string is immutable, everytime we try to change the state, we will end up creating a new object this might leads to memory leak problem.

To address this memory leak issue and problem ~~damage~~ Java introduced two more classes related to string and both of these two classes are mutable

* string builder
* string buffer.

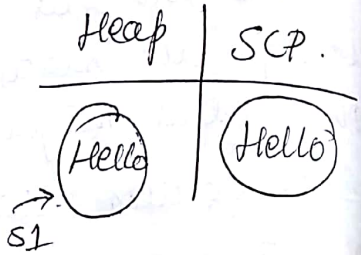
* Differences

String	String builder	String buffer.
→ It is fixed	→ It is final	→ It is final
→ It is immutable	→ Mutable	→ Mutable
→ It implements Comparable interface & array of string objects can be sorted	→ It doesnot implement comparable & hence array of string builder can't be formed	→ It doesnot implement Comparable interface & hence array of string buffer can't be formed.
→ Three methods of Object class has been overrided: toString(), hashCode(), equals().	→ only toString() method has been overrided	→ only toString() method has been overrided.

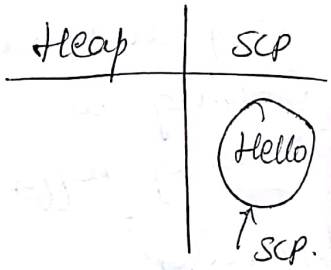
→ Methods are not synchronised.	→ Methods are not synchronized	→ Methods are synchronized
→ It is a thread safe	→ It is not a thread safe	→ It is a thread safe
→ Object can be created both by using new and without using new.	→ Object should be created using new only	→ Object should be created using new only.

* Constant pool (SCP) V/s Non-Constant pool (Heap)

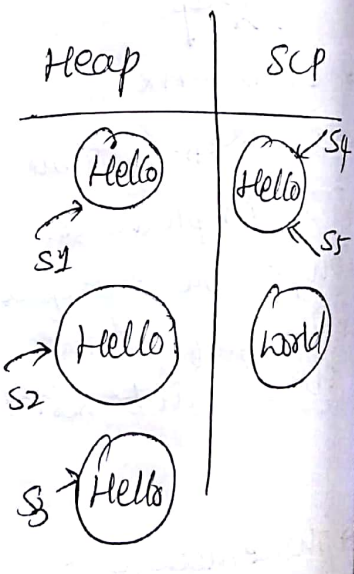
String s1 = new String ("Hello");



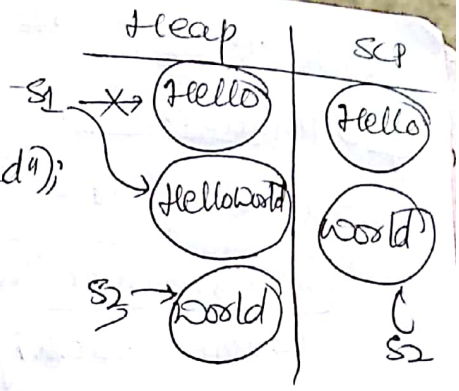
String s1 = "Hello";



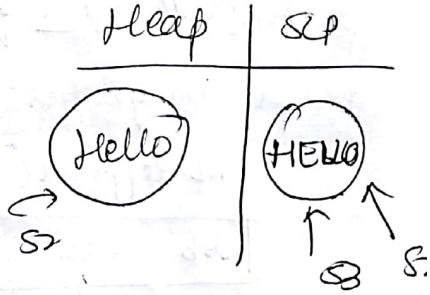
Eg: String s1 = new String ("Hello");
String s2 = new String ("Hello");
String s3 = new String ("World");
String s4 = "Hello";
String s5 = "Hello";
SOP (s4 == s5) // true



Eg: String S1 = new String ("Hello");
S1 = S1.concat ("World");
String S2 = "World";
String S3 = new String("World");



Eg: String S1 = "HELLO";
String S2 = S1.toLowerCase();
String S3 = S1.toUpperCase();
S1 == S2 ; // false
S1 == S3 ; // true



Whenever we create a String class object we cannot perform perform very changes in that object if we try to perform any changes, if there is change in the content then new object will be created with thoughs Changes if there is no change in the content then the existing object will be reused. This behaviour is known as Immutability.

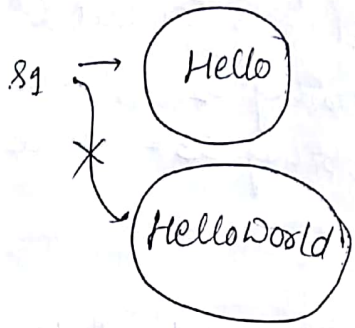
Eg1: String one = "Hello";
String two = "Hello";
if (one == two)
{
S.O.P("one == two"); ✓
}
else
{
S.O.P("one != two");
}

Eg2: String one = new String ("Hello");
String two = new String ("Hello");
if (one == two)
{
S.O.P("one == two");
}
else
{
SOP("one != two");
} ✓

Eg3: String buffer sf1 = new String buffer ("Hello");
String buffer sf2 = new String buffer ("Hello");
sf1.equals (sf2) ; // false

* final V/S Immutable.

Eg: ① Immutable & final.



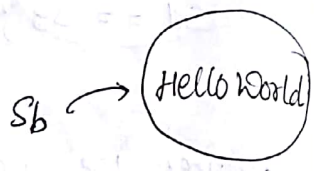

```
final String s1 = "Hello";
s1 = s1.concat("world");
```

↳ ∵ can't reassign

② Mutable and final

```
final StringBuffer sb = new StringBuffer("Hello");
sb.append("World");
sb = new StringBuffer("Hi");
```

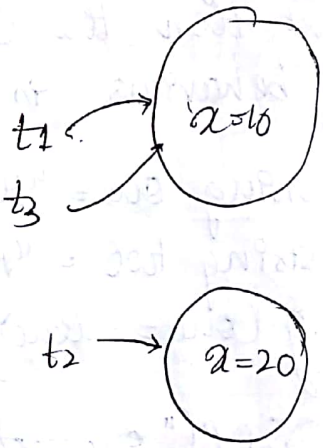
↓ Error



* Writing our own Immutable class.

Test.java

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



StringDemo6.java.

```
package jsp.string;
public class StringDemo6
{
  p s v m ( )
  {
    S.O.P(*****...***);
    Test t1 = new Test(10);
    Test t2 = t1.modify(20);//involves changes on the contents
    Test t3 = t1.modify(10);// doesn't involves changes on the contents

    if(t1==t2)
    {
      S.O.P("t1 & t2 ref to same obj");
    }
    else
    {
      S.O.P("t1 & t2 ref to diff obj");
    }
    if(t1==t3)
    {
      S.O.P("t1 & t3 ref to same obj");
    }
    else
    {
      S.O.P("t1 & t3 ref to diff obj");
    }
    S.O.P("*****...***");
  }
}
```

O/P: **********

t1 & t2 ref to diff obj

t1 & t3 ref to same obj

****

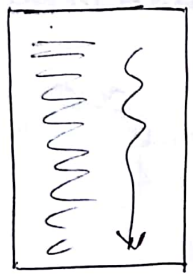
* Exceptions

→ What is an error ?.

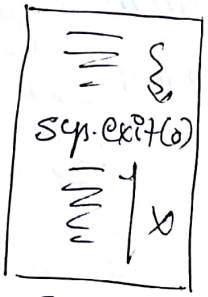
* Error is something which might occur either during compile time or run-time

* Compile time errors are thoughs errors which are caught by the compiler during compilation.

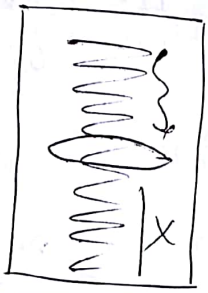
* Run-time errors are thoughs errors which occur during run-time.



Normal termination



Forceful termination



Abnormal termination Exception

Eg: ← PLAN →

6am → 7:30am —10min→ 7:40 → 7:50 ←1hr→ 8.50 → Reaching Yelahanka 9:00am

↓ wakeup — ↓ leave home — ↓ reach bus stand — ↓ get bus — ↓ Reach venue

Alternatives

* Bus Not coming Exp.
* Bus stuck in Traffic Exp.
* Bus Break down Exp.

→ Exception is an unexpected unwanted 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

26/10/17

* Default Exception Handler:-

When there is an exception in a program, and if user had not written any code to handle the exception, then JVM handles the exception, by calling default exception handler.

When there is an exception in a program, JVM will create an object of respective type of exception and throws. If user has not written any code to catch that exception object, then default exception handler will handle.

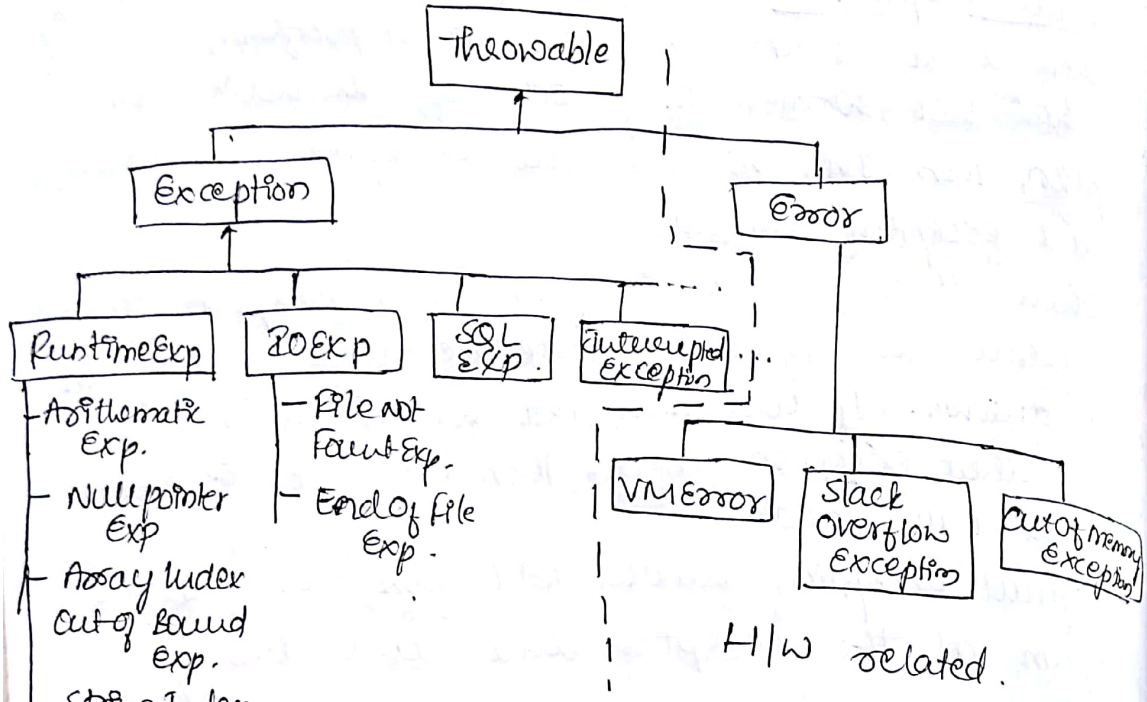
Default exception handler will first terminate the program at the exception line and display the information about the exception to the end user.

NOTE:
In java every exception is a class.

Eg: Package jsp.exceptions;
public class ExceptionDemo1
{
P S V M (String[] args)
{
SOP(" M M S");
int x=10;
int y=0;
SOP("DIV = "+x/y); // exp line
SOP(" M M ended"); // It doesnot execute
}
}

O/P: M M S
Exp in thread "main" java.lang.ArithmeticException: / by zero
at jsp.exceptions.ExceptionDemo1.main(ExceptionDemo1.java:12)

* Exception Hierarchy in Java.



* Handles.

1. Try.
2. Catch.
3. Throw
4. Throws.
5. Finally.

* Try/Catch Block

→ Try block will contain only those lines of code which might cause an exception.

→ Once the exception occurs in the try block the corresponding exception object will be created and thrown out of try block.

→ User should define, the corresponding catch block to catch the exception thrown from the try block.

→ Once the controller comes out of the try block, it will never go back to try block again. Hence it is

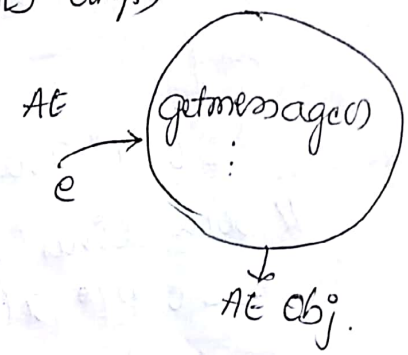
recommended to keep as many minimum number of lines as possible in the try block.

27/10/17

ExceptionDemo1

```
package jsp.exceptions;
public class ExceptionDemo1
{
  public static void main(String[] args)
  {
    S.O.P ("M M S");
    int x = 10;
    int y = 0;

    try
    {
      S.O.P ("entering try block");
      S.O.P (x/y); // exp line
      S.O.P("exiting try block"); // this will not execute.
    }
    catch (ArithmeticException e)
    {
      S.O.P("entering the catch block");
      S.O.P(e.getMessage());
      S.O.P("exiting catch block");
    }
    S.O.P ("Main method ended");
  }
}
```



o/p: M M S
entering try block.
entering the catch block
/ by zero
exiting catch block.
Main method ended.

Eg2: package jsp.exception

```
public class ExceptionDemo2.
{
psvm ( )
{
  S.O.P ("MMS");
  int x=10;
  int y=0;
 Int[] a = new int[5];
 try
 {
  S.O.P("Inside try block");
// both lines are exception but only 1st occurs.
  S.O.P(x/y);
  a[5] =100;
  S.O.P("exit try block");
 }
 Catch (ArithmaticException e)
  {
    S.O.P ("Inside AE");
    S.O.P(e.getmessage());
  }
 Catch (ArrayOutOfBoundsException e)
  {
    S.O.P("Inside AOFBE");
    S.O.PC" e.getmessage()); // 5
  }
    S.O.P("MME");
 }
}
```

Op: MMS
Inside try block
Inside AE
/ by zero
MME

eg3:

```
try
{
}
catch (AE e)
{
}
catch (NPE e)
{
}
catch (AIOBE e)
{
}
```

Specialized catch

```
try
{
}
catch (Exception e)
{
}
```

Generalized block for all exception

Note:
You can write specialized and generalized catch blocks together for a single try block provided the order should be first specialized and then generalized

```
try
{
  SOP(10/0)
}
catch (AE e)
{
}
catch (Exception e)
{
}
```

```
try
{
  SOP(10/0)
}
catch (Exception e)
{
}
catch (AE e)
{
}
```

error

* Multiple try - catch Blocks:

Eg:
```
public class Demo3
{
    p s v m ( )
    {
        S.O.P(" M M S");
        int x=10;
        int y=0;
        int [] a = new int[5];
        try
        {
            S.O.P (x/y);
        }
        catch (ArithmaticException e)
        {
            S.O.P(" Inside AE");
            S.O.P( e. getMessage());
        }
        try
        {
            S.O.P( a[5] = 100;
        }
        catch (ArrayIndexOutOfBoundsException e)
        {
            S.O.P(" Inside AIOFB");
            S.O.P(e. getmessage());
        }
        S.O.P("M M E");
    }
}
```

O/P: M M S
Inside AE
/ by zero
Inside AIOFB
S

① try
{
}
catch(-)
{
}
✓

② try
{
}
catch(-)
{
}
catch(-)
{
}
✓

③ try
{
}
catch(-)
{
}
try
{
}
catch(-)
{
}
✓

④ try
{
}
✗
Only try not allowed.

⑤ try
{
}
catch(-)
{
}
finally(-)
{
}
✓

⑥ try
{
}
finally
{
}
✓

⑦ try
{
}
catch(-)
{
}
finally(-)
{
}
finally(-)
{
}
✗

⑧ try
{
}
try
{
}
catch(-)
{
}
{
}
catch(-)
{
}
✓

⑨ try
{
}
try
{
}
catch
{
}
catch
{
}
✗

⑩ try
{
}
SOP(-); ✗
catch(-)
{
}
✗

* Throw keyword.

→ Throw is a keyword used to throw both checked and unchecked exception object.

→ Throw will throw only though exception objects which has the properties of throwable type

i.e., throw new AE(); } valid only if these
throw new test(); } objects are having
throw new Demo(); } properties of throwable type.

→ Using throw we can throw only one exception object at a time. i.e., `throw e1, e2;` is invalid.

Eg: ① How to throw an exception explicitly.

```
Class ExceptionDemo4
{
Public static void main (String [] args)
{ S.O.P ("M M S");
  try
  {
    throw new Arithmatic Exception (" stupid don't divide by zero");
  }
  catch (Arithmatic exception e)
  {
    S.O.P( e.getmessage());
  }
    S.O.P ("M M E");
  }
}
```

※ ② try
{
S.O.P("Hi"); → ①
S.O.P(10/0);
S.O.P("Bye"); ×
}
Catch (AE e)
{
S.O.P ("Exp Caught"); → ②.
}

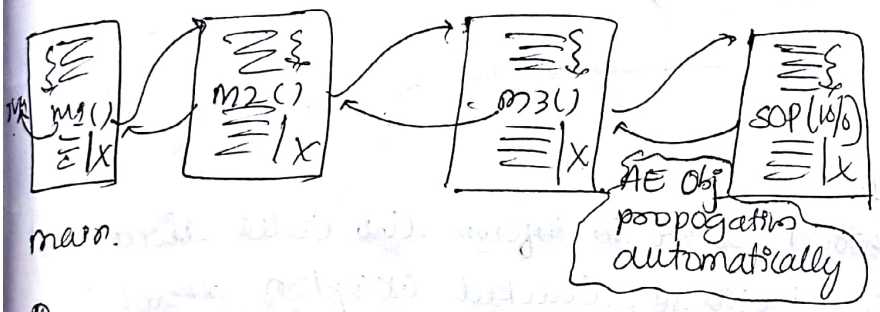
try
{
S.O.P ("Hi");
throw new AE();
S.O.P ("Bye");
}
Catch (AE e)
{
S.O.P ("Exp Caught");
}

Compile time error

try
{
new AE();
}
catch (AE e)
{
S.O.P("exp caught");
}

Exception Propogation

Whenever exception occurs, JVM will create any object of the respective exception type and throws it. If user had not written any code to handle that exception, that exception object will automatically propogate back to its caller. This automatic propogation capability is there only for unchecked exceptions (checked exception objects will not propogate automatically)



Eg: Package jsp.exceptions;
public class ExceptionDemo
{
PSVM("MMS");
{
S.O.P("M.M.S");
m1();
S.O.P("M M E");
}
public static void m1()
{
S.O.P("M1() method started");
m2();
S.O.P("m1() method ended");
}

```
public static void m2()
{
  S.O.P("m2() ME");
  m3();
  S.O.P("m2() ME");
}
Public static void m3()
{S.O.P("m3() MS");
  try
  {
    S.O.P(10/0);
  }
  Catch(ArithmaticException e)
  {
    S.O.P("exception handled");
  }
  S.O.P("m3() ME");
}
```

*Throws keyword.

→ It is a keyword used to inform the caller about the checked exception. Because, checked exception do not have the capability of propogating through the callee by itself.

→ It must be used in the method declaration.

→ Using throws, we can inform multiple checked exceptions to the caller.

```
public void m1() throws SQLExp, IoExp.
{
}
```

Eg① package jsp.exception

```
import java.SQLException;
public class InsertQuery
{
  Public void insertData(String query) throws
                                        SQLExpection
```

```
S.O.P("InsertData method started");
if (query != null)
// write code to insert data into the query
  S.O.P("Inserted successfully");
}
else
{
  throw new SQLException();
}
S.O.P("InsertData method ended");
}
```

Main class

```
Package jsp.exceptions;
import java.sql.SQLException;
public class ExceptionDemo6
{
  P S V M (String[] args) throws SQLException
  {
    S.O.P("M M S");
    String query=" insert into emp values (10, 'Durga', 70);
    Insertquery iq= new Insertquery ();
    iq.insertData (null);
    S.O.P("M M E");
  }
}
```

```
Public class Printalumbers
{
  Public void print()
  {
    S.O.P("print() method started");
    for (int i= 1; i<=10; i++)
    {
      S.O.P("i="+i);
      try
      {
        Thread.sleep (1000);
```

```
catch (InterruptedException e)
  {
    e.printStackTrace();
  }
}
```

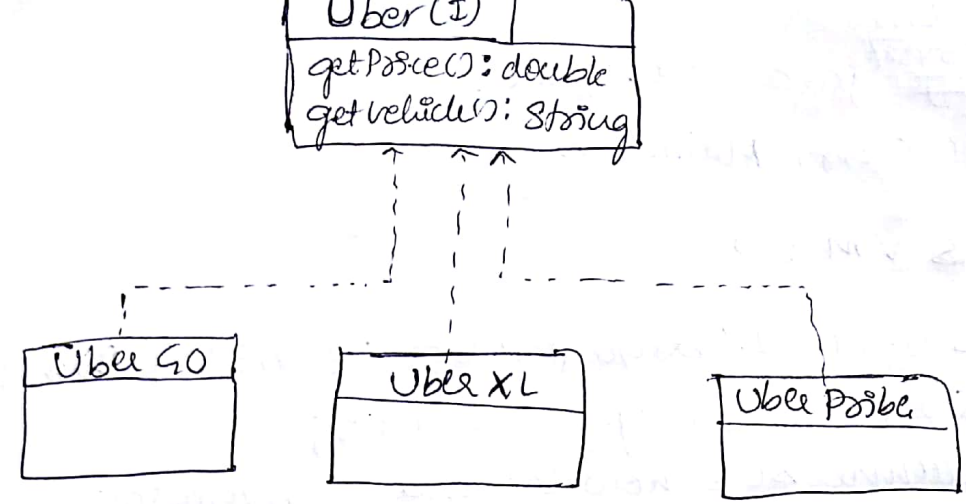
main

```
package jsp.exceptions;
public class ExceptionDemo7
{
  public static void main (String[] args)
  {
    S.O.P("M M S");
    PrintNumbers pn = new PrintNumbers();
    pn.print();
    S.O.P("M M E");
  }
}
```

*ABSTRACTION :

→ It is the process of hiding the implementation of object and showing only the functionality of an object.
→ Abstraction can be achieved by following the below the 3 steps.

(a) Identify all the common properties of an object
(b) Keep them in the Interface.
(b) Provide multiple implementation class for that interface.
(c) Create an interface type reference and make it to refer to any of its implementation class object



Eg:

```
Interface Uber.
{
  double getPrice();
  String getVehicle;
}

Package jsp.Uber.
Public class UberGo implements Uber
{
  Public double getPrice()
  {
    return 250.00;
  }
  Public String getVehicle()
  {
    return "micro";
  }
}
```

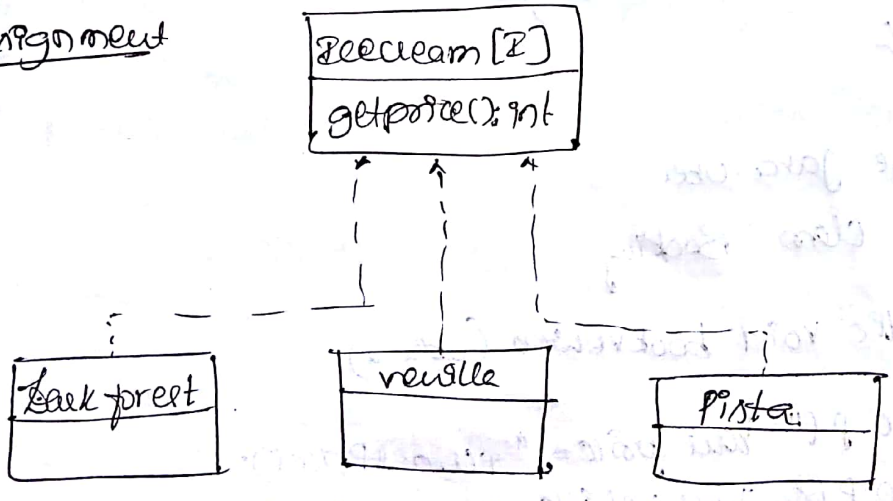
Uber for UberXL & Uber prime

```
Package java.Uber
Public class Booking
{
  Public void bookVehicle(Uber u)
  {
    S.O.P("Your price=" + u.getPrice());
    S.O.P("Your vehicle=" + u.getVehicle());
  }
}
```

Main class.

```
import java.util.Scanner;
Public class MainClass
{
  P.S.V.M()
  {
    S.O.P("1.UberGo \n2.UberXL\n3.UberPrime\n4.Exit");
    S.O.P("Enter your choice");
    Scanner sc = new Scanner(System.in);
    int ch = sc.nextInt();
    Booking b = new Booking();
    switch(ch)
    {
      Case1: b.bookVehicle(new UberGo());
             break;
      Case 2: b.bookVehicle(new UberXL());
             break;
      Case 3: b.bookVehicle(new UberPrime());
             break;
      Case 4: System.exit(0);
      default: S.O.P("Invalid option");
    }
  }
}
```

Assignment



29/10/17

* Differences b/w Checked Exception & Unchecked Exception

Checked	Unchecked
→ known at compile time but occurs at Run-time	→ known and occured at Run-time.
→ It don't have the capability of propogating to the caller by itself.	→ It has the capability of propogating to the caller by itself.
→ Throws keyword is required to inform the caller about the checked exception.	→ Throws keyword is not required.
→ Compiler forces to provide alternative during compilation.	→ Compiler doesn't force
→ Ex: Exp, IOExp, InterruptExp SQLException.	→ Runtime and its sub-class.

Throw	Throws
→ It is used to throw both checked and unchecked exception objects which has the properties of throwable time.	→ It is used to inform the caller about the checked exception.
→ It should be used inside the method defination	→ It should be used on the method declaration.
→ Using throw we can throw only one exception object at a time.	→ Using throws, we can inform caller about multiple checked exception object.
→ throw e_1, e_2;	→ Public void m1() throws SQLExp, Interrupted Exp, IOExp.

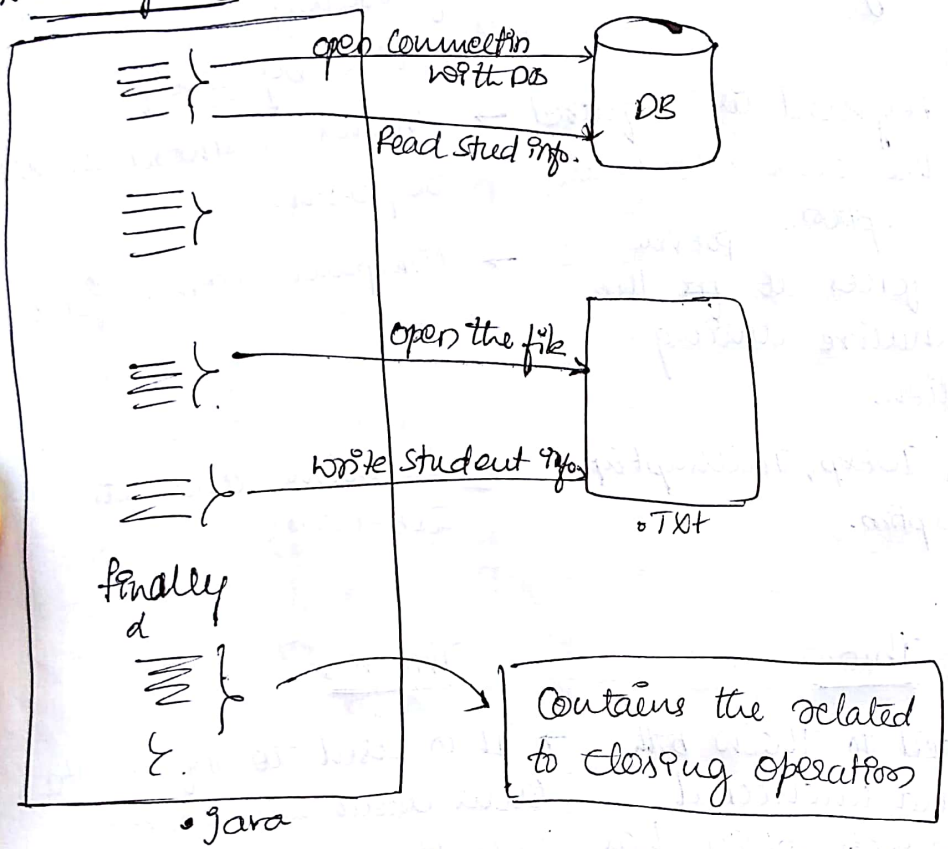
Throw

→ It is a keyword used to throw exception objects.

Throwable

→ It is the super most class in the exception hierarchy, all checked & unchecked exceptions are inherited from this cl

* Finally Block.



Finally Block contains only thoughs lines of code which generally performing closing operations. like, closing DB connection, closing the opened file, closing the opensection, closing the opened terminal etc.

```
try
{
SOP(10/0);
}
Catch(AE e)
{
}
Finally
{
}
```

```
try
{
SOP(10/5)
}
Catch(AE e)
{
}
finally
{
}
```

```
try
{
SOP(10/0)
}
finally
{
}
```

JVM handles.

```
try
{
 return;
}
finally
{
 ==| ✓
}
```

```
try
{
 System.exit(0);
}
finally
{
 ==| ✗
}
```

* Writing customer unchecked exception.

Ex: Package . jsp . exceptions;

```
public class InsufficientBalanceException extends
                RuntimeException.
{
  private String msg;
  public InsufficientBalanceException (String msg)
  {
    this.msg= msg;
  }
  public String getmessage()
  {
    return msg;
  }
}
```

```
package jsp.exceptions;
public class Transaction
{ public void withDraw(double amtBal, double amtWithDraw)
  {
    S.O.P("WithDraw method Started");
    if (amtBal > amtWithDraw)
    {
      S.O.P("WithDraw Successful");
      amtBal = amtBal - amtWithDraw;
    }
    else
    { try
      {
        throw new InsufficientBalanceException("Maintain
                                                Balance");
```

```
Catch (InsufficientBalance exception e)
{
    S.O.P (e.getMessage());
}
}
S.O.P ("Balance = " + amtBal);
S.O.P("withdraw method ended");
}

Public class exceptionDemo9
{
  P.S.V.M (String[] args)
  {
    S.O.P ("Main method started");
    double amtBal = 50000.00;
    double amtWithDraw = 60000.00;
    Transaction t = new Transaction();
    t.withDraw (amtBal, amtWithDraw);
    S.O.P("main method ended");
  }
}
```

* Writing custom checked exceptions

Eligibility for voting

```
package jsp.exceptions.
public class VotingNotEligibleException extends Exc
{
  private String msg;
  public VotingNotEligibleException (String msg)
  {
    this.msg = msg;
  }
  public String getMessage()
  {
    return msg;
  }
```

* Drawback of arrays.

→ Arrays are fixed in size.

→ Arrays store only homogenous data.

→ Arrays always demands for continuous memory location, and hence it doesnot utilize the memory effectively

→ No readymade methods are available. Because arrays doesnot implemented on any standard data structure.

TO overcome all the above drawbacks of arrays, we will go for collection.

* Difference between array and collection.

Array	collections
→ Fixed in size	→ Growable in nature.
→ Stores only homogeneous type of data.	→ It stores both homogeneous and heterogeneous data.
→ No readymade methods are available.	→ Lot of readymade methods are available Because, every collection is implemented on some standard data structure.
→ It utilizes the memory ineffectively	→ It utilizes the memory effectively.
→ Arrays are better in performance	→ Collections are better in memory utilization.
→ We can store both primitive values and no primitive values, by creating respective type of array	→ We can store only non primitive types (objects) in the collection.

* COLLECTION FRAMEWORK LIBRARY. (CFL)

It defins the set of classes and interfaces to overcome the drawbacks of arrays. There set of classes and interfaces are available in Java.util package.

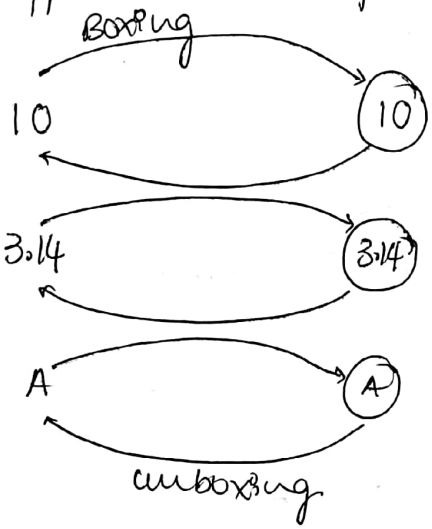
In C++, containes topic is renamed as collection in Java. Standerd template Library (STL) in C++ and CFL in Java.

* WRAPPER CLASS

30/10/17

→ These are used to convert primitive type to class type and class type to primitive type.
→ Each primitive data type has its corresponding wrapper class.
→ All wrapper classes are available in java.lang.
→ All wrapper classes are final.
→ All wrapper classes override to_String(), hashcode() and equals(Object obj).
→ All wrapper classes implement comparable interface

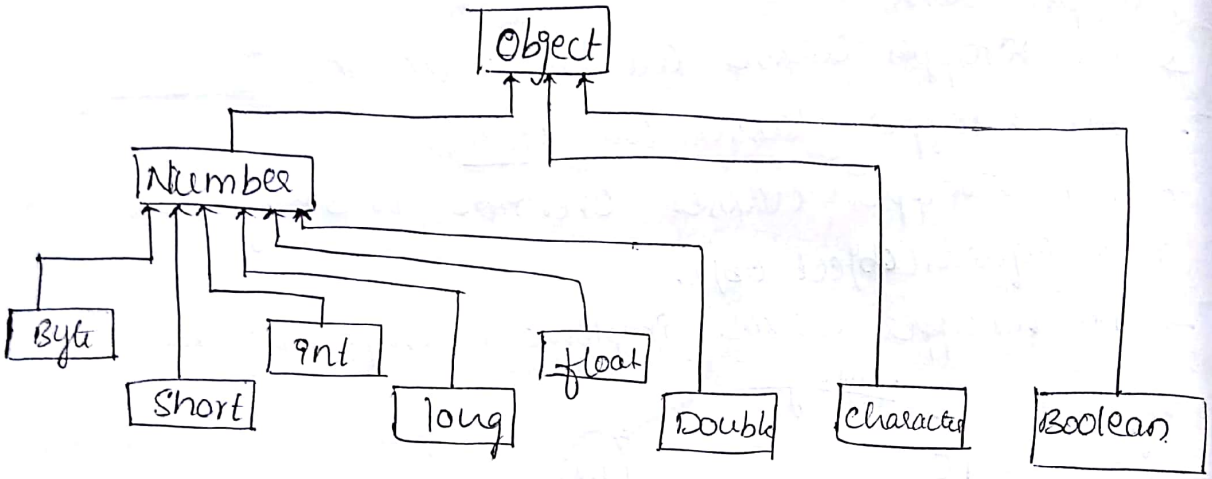
Eg:



Primitive Data type		Wrapper class
byte	→	Byte.java
short	→	Short.java
int	→	Integer.java
long	→	Long.java
float	→	Float.java
double	→	Double.java
char	→	Character.java
boolean	→	Boolean.java

→ Converting primitive to class called Boxing. This can be done either implicitly by compiler (Auto Boxing) or explicitly by the user (Manual/explicit Boxing).

→ Converting class type to Primitive type is known as UnBoxing. This can also be done either implicitly by the compiler or explicitly by the user (Manual Unboxing) (Auto unboxing)



① Can we ~~Override~~ ~~change the~~ change the return type while overriding?
NO, if the return type is primitive. However Overriding class type return type can be changed, that too, Covariant return type but not contravariant. R.T.

Eg:
```
Class A
{
  public void m1()
  {
    ==
  }
}
Class B extends A
{
  public int m1()
  {
    ==
  }
}
```

Cannot Change primitive type R.T.

```
Class A
{
  public Object m1()
  {
    ==
    return new Object();
  }
}
Class B extends A
{
  public String m1()
  {
    ==
    return new String();
  }
}
```

Covariant R.T.

```
class A
{
  public ~~void~~ String m1()
  {
    ==========
    return new String();
  }
}
class B extends A
{
  public Object m1()
  {
    ==========
    return new Object();
  }
}
```

Contravariant : R.T

Eg2: * Test t = 10;
Int i = new Test(); } ✗

Int x = 10;

Integer ptr1 = 10; // Auto Boxing
Integer ptr2 = Integer.valueOf(x);

Boxing

explicit Boxing.
Static method

ptr1 → 10
ptr2 → 10

* Int m = ptr1; // Auto Unboxing.
Int n = ptr2.intValue(); // Explicit Unboxing.

Non static method.

SOP (m); //10
SOP (n); //10.

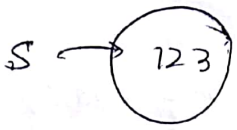
SOP (itr1);
SOP (itr2); } 10

toString() of Integer class

* String s = "123";

int m = s; // X

s.o.p(s); // 123
↳ this string.



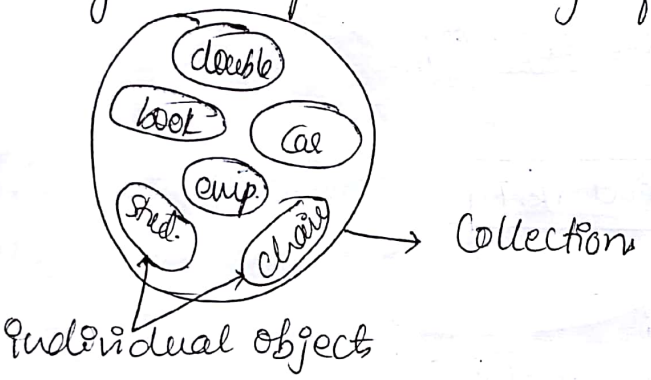
int n = Integer.parseInt(s);

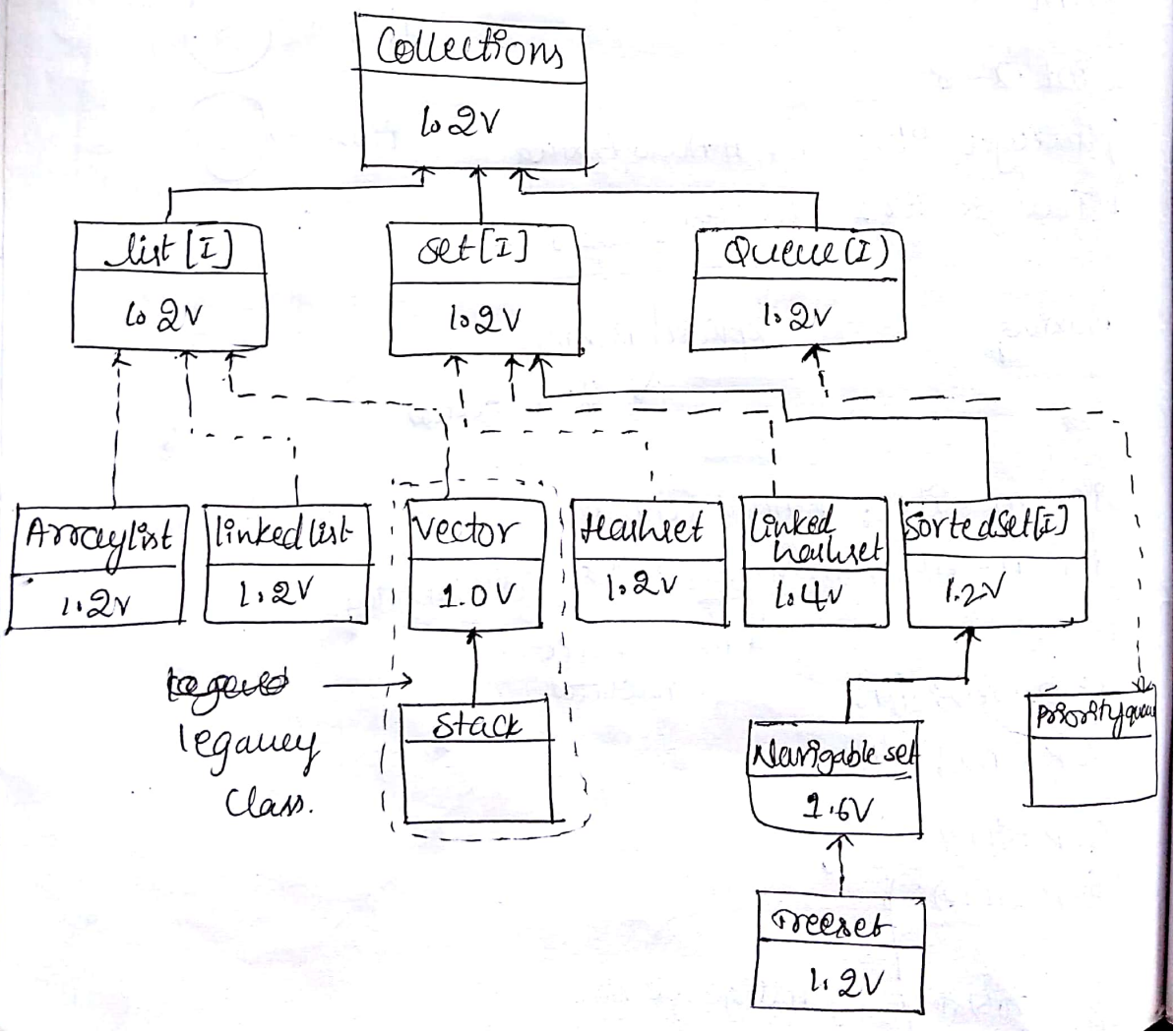
s.o.p(n); // 123
↳ It is number.

Collections

31/11/17

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





Collection (I)

It acts as a root interface in the entire Collection framework library. It is introduced in collection interface defines the most commonly used method which are applicable for any type of collection.

* Collection interface methods:-

→ public boolean add (Object obj)
→ public boolean addAll (Collection c)
→ public boolean removeAll (Collection c)
→ public boolean contains (Collection c)
→ public boolean containsAll (Collection c)
→ public boolean retainAll (Collection c)
→ public void clear()
→ public int size()
→ public boolean isEmpty()
→ public ~~boolean~~ Object() to Array()
→ public Iterator Iterator ()

* List (I):

→ Sub Interface of Collection introduced in 1.2V. If you want to represent a group of individual objects a single entity where duplicates are allowed and insertion order is preserved, then we will go for List.

* List specific methods:-

→ void add (int index, object obj)
→ boolean addAll (int index, Collection c)
→ object get (int pos)
→ int indexOf (Object obj)
→ int lastIndexOf (object obj)
→ object set (int index, object obj) // replace

→ ListIterator listIterator()
→ Object remove(int index)

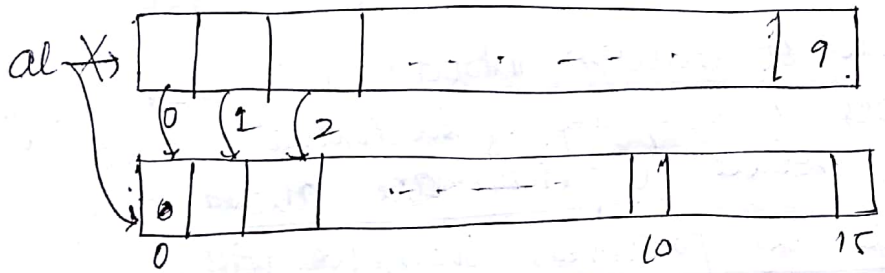
* Arrays List:

→ It is a implementation class version of List introduced on 1.2V.
→ The underlined datastructure is resizable or growable array.
→ Duplicate objects are allowed.
→ Insertion order is preserved.
→ Heterogenous objects are allowed.
→ Null insertion is possible.
→ It implements serializable, clonable and Random access interface.

* Constructors Of Array List

→ ArrayList al= new ArrayList();
*creates an empty array list object with the default initial capacity 10.
*Once the arraylist reaches it maximum capacity it creates a new arraylist object with the new capacity

New capacity= Current Capacity * 3/2 + 1

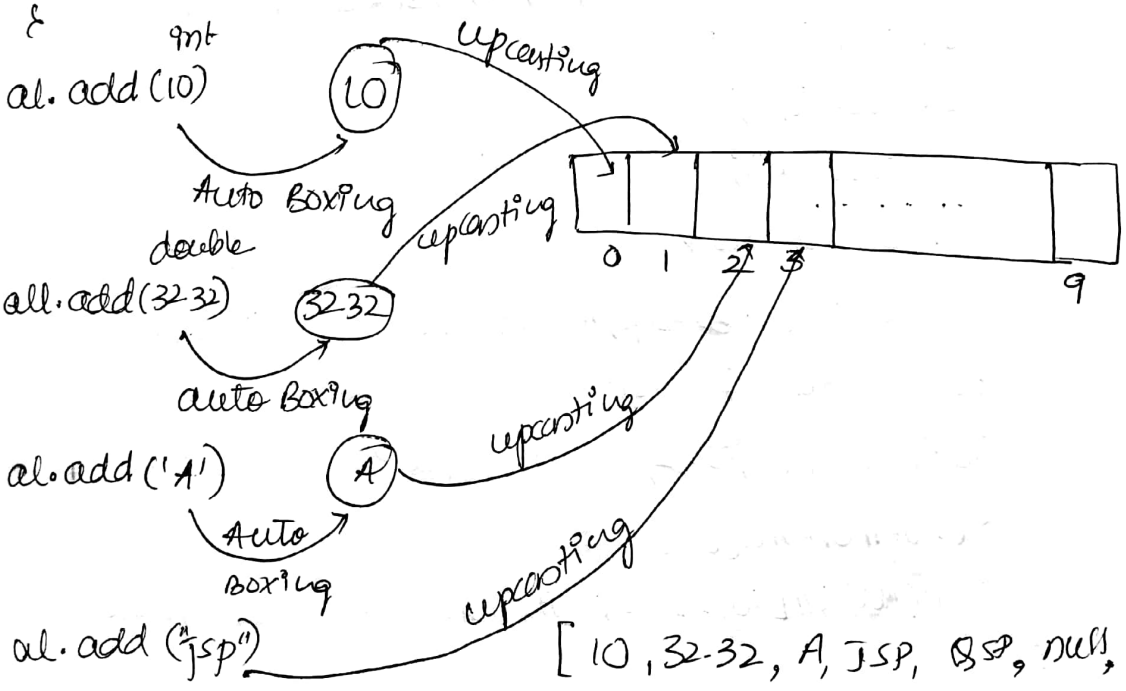


→ Arraylist al= new Arraylist(int initialCapacity)
* creates an empty arraylist object with the specified initial capacity.
→ Arraylist al= new Arraylist(Collection c)
* Converts the given collection into arraylist equivalent.

Eg: Package jsp.jcf; → import java.io.Serializable;
import java.util.ArrayList;
Public class ArrayListDemo1.java; | util.RandomAccess;

```
{
  p s v m(String[] args)
  {
    ArrayList al = new ArrayList();
    S.O.P("Before = " +al.size());
    al.add(10);
    al.add(32.32);
    al.add('A');
    al.add("JSP");
    al.add("JSP");
    al.add(null);
    al.add(10);
    al.add(20);
    al.add(40);

    S.O.P(al);
    S.O.P(al instead of cloneable);
    S.O.P(al instead of serializable);
    S.O.P(al instead of RandomAccess);
    S.O.P("After = "+al.size());
  }
}
```



Eg2:

```
Package jsp.jcf;
import
import
import
public s.v m(String[] args)
{
  ArrayList al = new ArrayList();
  ArrayList al1= new ArrayList();
   al. add(10);
   al. add(32.32);
   al. add('A');
   al. add("JSP");
   al. add("QSP");
   al. add(null);
   al. add(10);
   al. add(20);
   al. add(40);
   S.O.P(al); // [10,32.32, A, JSP, QSP, null, 10, 20, 40]
   al. add(2, 'Z');
   S.O.P(al); // [10, 32.32, Z, A, JSP, QSP, null, 10, 20, 40]
   al. set(6, 1000);
   S.O.P(al); // [10, 32.32, Z, A, JSP, QSP, 1000, 10, 20, 40]
   al1. add('P');
   al1. add('Q');
   al1. add('R');
    al. addAll(al1);
   S.O.P(al); // [10, 3232, Z, A, JSP, QSP, 1000, 10, 20, 40, P, Q, R]
   al. addAll(4, al1);
   S.O.P(al); // [10, 32.32, Z, A, P, Q, R, QSP, 1000, 10, 20, 40, P, Q, R]
   al. removeAll(al1);
   S.O.P(al); // [10, 32.32, Z, A, JSP, QSP, 1000, 10, 20, 40]
}
```

* Advantages of Arraylist

→ Since arraylist implements random access interface, retrival operation is faster. Hence if the frequent operation is retrival operation, arraylist is recommended.

Disadvantage

→ Insertion or deletion in b/w the arraylist internally involves lot of shift operations, which might brings down the performance of the application. Hence if the frequent operation is insertion or deletion in b/w, arraylist is not recommended.

02/11/17

* In an arraylist of heterogenous object, write a program to display only integer object

```
package jsp.jcf;
import java.util.ArrayList;
public class ArrayListDemo1
{
    P S V M ( String[] args)
    {
      S.O.P ("****.....****");
      ArrayList al = new ArrayList();
       al.add(10);
       al.add(13);
       al.add("JSP");
       al.add("asp");
       al.add('A');
       al.add(543.34);
       al.add(68);
       al.add(54);
       al.add(null);
    for(int i=0; i<al.size(); i++)
    {
      Object obj = al.get(i);
```

```
if (obj instanceof Integer)// checking if obj is integer
{
  Integer itr = (Integer) obj; // downcasting.
  S.O.P(obj); // toString() of Integer
}
{
 S.O.P("***.....***");
}
```

o/p: 10 13 65 54.

* write a program to count the number of string objects on an arraylist.

soln:-
```
int count=0;
for(int i=0; i<al.size(); i++)
 {
   Object obj = al.get(i);
   if(obj instanceof String)
   {
     count++;
   }
 }
```

* display only even integer numbers.

soln
```
for(int i=0; i<al.size(); i++)
 {
   Object obj = al.get(i);
   if(obj instanceof Integer)
   {
    Integer itr = (Integer) obj;
     if(itr%2 ==0) // autounboxing of ref var itr
     {
       S.O.P(obj);
     }          (or obj also can be used.)
   }
 }
```

* Display only if object is JSP strings. from array

```
for (int i=0; i<al.size(); i++)
{
  Object obj = al.get(i);
  if (obj instanceof String)
  {
    String str = (String) obj;
    if (str.equals("Hello"))
    {
      S.O.P(str);
    }
  }
}
```

* Write a program to replace all odd integer numbers with even numbers and remove the remaining objects from array list collection.

Soln

```
for (int i=0; i<al.size(); i++
{
  Object obj = al.get(i);
  if (obj instanceof Integer)
  {
    Integer ito = (Integer) obj;
    if (ito%2!=0)
    {
      al.set(i,100);
    }
  }
  else
  {
    al.removeAll(i);
  }
}
```

Note:
Here backtracking happens hence below number of odd retains at it is.

* Hold a customised object (student) of a collection

Eg: package jsp.jcf;

```
public class student
{
  String name;
  int id;
  double marks;
  public student(string name, int id, double marks)
  {
    this.name = name;
    this.id = id;
    this.marks = marks;
  }
  public string tostring()
  {
    return this.name "\t\t" +this.id+ "\t\t" +this.marks
  }
}
```

06/11/17 Employee.java

```
package jsp.jcf;
public class Employee
{
  string name;
  int id;
  double sal;
  public Employee(string name, int id, double sal)
  {
    this.name= name;
    this.id  =id;
    this.sal = sal;
  }
  public string tostring()
```

ArrayListDemo6.java

```
package jsp.jcf;
public class ArrayListDemo6
{
  Public . S. V. m (String[] args)
   {
    ArrayList al = new ArrayList();
    al. add (new Student("Durga", 10, 76.45);
    al. add (new Student ("Durga", 20, 80.6);
    al.add (new Student ("Sweepna", 30, 90.2);
    al. add (new employee ("Shanthi", 100, 22000");
    al. add (new employee (" Ravi", 200, 23000);
    S.O.P ("Name\t\tId\t\t marks");
    S.O.P ("-----------");
    for (int i=0, i< al.size(); i++)
     {
      object obj = al.get(i);
      if (obj instanceof student)
        {
          student s = (student) obj
          S.O.P(s);
        }
      else if (obj instanceof employee)
        {
          employee e = (employee) obj;
          S.O.P(e);
        }
     }
   }
}
```

* In an arraylist of Student and employee object, write a program to display only thoughs students whose marks ≥ 60.

```
for (i=0; i< al.size(); i++)
  {
    Object obj = al.get(i);
      if ( obj instanceof Student)
        {
          Student s. (Student) obj;
            if (s.marks > 60)
              {
                S.O.P (s);
              }
        }
  }.
```

* In an arraylist of student and employee object, write a program to display ~~name~~ all the employees whose name starts with D.

Soln

```
for (i=0; i< al.size(); i++)
  {
    Object obj = al.get(i);
      if (obj instanceof ~~Student~~ employee)
        { employee e
          ~~student~~ s. employee) obj;
            if (e.name.startsWith("D"));
              {
                S.O.P(e);
              }
        }
  }
```

* In an arraylist of student object, write a program to display all the students whose marks > 60 and the name starts with either M or B.

Soln

```
for (i=0 ; i< al.size(); i++)
  {
    Object obj = al.get(i);
```

```
if (obj instanceof Student)
{
    Student s = (Student) obj;
    if (s.marks > 60 && (s.name.startswith("A") ||
                         s.name.startswith("B")))
    {
        s.DP(s)
    }
}
```

Assignment

1. In an arraylist of heterogeneous objects write a prog to display all Kannada movies.
2. In an arraylist of heterogeneous objects write a prog to display all the books whose price is > 250 and < 500.
3. In an arraylist of heterogeneous objects write a program to display all the cars whose color is white. and the price < 50,00,00 (c.color.equals("w

* Linked List:

- It is an ~~indexed~~ implementation class of List introduced in 1.2V.
- The underlined data structure is Doubly Linked List
- Heterogeneous objects are allowed.
- Duplicates are allowed.
- Insertion order is preserved.
- Null insertion is possible.
- It implements serializable interface and clonable interface. But not random access.

* Constructors of LinkedList

① LinkedList l = new LinkedList()

It creates an empty linkedlist object.

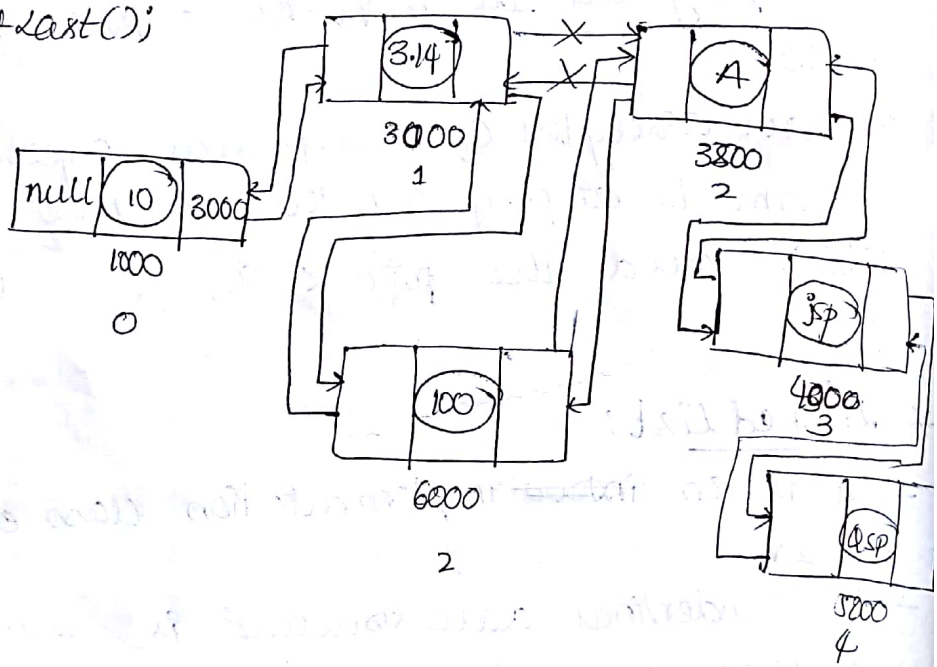
② LinkedList l = new LinkedList(Collection c)

Converts the given collection into linkedlist equivalent

* LinkedList Specific methods

① void addFirst(Object obj)
② void addLast(Object obj)
③ Object removeFirst();
④ Object removeLast();
⑤ Object getFirst();
⑥ Object getLast();

```
l.add(10);
l.add(3.14);
l.add('A');
l.add("jsp");
s.o.p(l);
l.add("qsp");
l.add(2,100);
```



08/11/17

Eg)

```
Package jsp.jcf.
import java.util.LinkedList;
public class LinkedListDemo1
{
    P S V M (String[] args)
    {
        LinkedList l = new LinkedList();
        l.add(10);
        l.add(432.43);
```

```
l.add("QSP");
l.add(Null);
l.add(10);
l.add(32);
l.add(55);
S.O.P(l); //[10, 432.42,
l.add(5, "Hello");
S.O.P(l);
l.addFirst(100);
l.addLast(1000);
S.O.P(l);
S.O.P(l.getFirst()); //100
S.O.P(l.getLast());
l.removeFirst();
l.removeLast();
S.O.P(l);
}
}
```

* Differences.

ArrayList	LinkedList
→ Underlined datastructure is growable / Resizable array	→ The underlined datastructure is doubly LinkedList
→ It implements Random access	→ It doesn't implement random access.
→ Objects will be stored in consecutive memory location	→ Objects will not be stored ~~under~~ in consecutive M.L.
→ Retrival operation is faster	→ Insertion / deletion operation is faster.

*Set(I):→ Subinterface of Collection introduced in 1.2

→ If you want to represent a group of individual object as a single entity where duplicates are not allowed and insertion order is not preserved then we will go for set.

*Differences

List(I)	Set(I)
→ Insertion order are preserved	→ Insertion order not preserved.
→ Duplication is allowed	→ Duplication is not allowed.

① *Hash Set:-

→ It is an implementation class of set introduced in 1.2v.
→ The underlined data structure is HASH TABLE.
→ Duplicate objects are not allowed.
→ Insertion order is not preserved. Objects will be inserted based on the hashcode.
→ Null insertion is possible.
→ If the frequent operation is search operation then it is recommended to use any of the hashing related Collection.

NOTE:-

Hashset doesnot allow duplicates. If we try to add any duplicate objects, we will not get any error a exception instead add method simply returns false.

*Constructor of Hash set

① Hashset h = new Hashset();

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

Fill Ratio :- It is also known as load factor which indicates the criteria to 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 f

Creates an empty hashset object with the specified initial capacity and the specified fill ratio.

④ Hashset h = new Hashset(Collection c)

Converts the given collection into hashset equivalent.

Eg: HashsetDemo1.java

```
package jsp.jcf;
import java.io.Serializable;
import java.util.Hashset;
import java.util.RandomAccess;

public class HashsetDemo1
{
    p s v m(String[] args)
    {
        Hashset h = new Hashset();
        S.O.P("Before size = "+h.size()); //0
        h.add(10);
        h.add(20);
        h.add(A);
```

```
h.add ('QSP');
h.add (343.42);
h.add (30);
S.O.P (h.add(10)); // false.
S.O.P(h); // [10,20,'A','QSP',343.42,30]
S.O.P("After size= "+h.size());  // 4
S.O.P(h instanceof Cloneable);
S.O.P(h instanceof Serializable);
S.O.P(h instanceof RandomAccess);
}
}
```

Cursors

→ Cursors are used to retrive the objects from the collection one by one.

→ Java provides 3 types of cursors

① Enumeration (I):- Stack & Vector (legacy), only Read operation, two method

② Iteration (I) :- Any type of Collection, Universal cursor, Read/Remove, 3 methods, unidirectional.

③ ListIterator (I): Powerfull cursor, Bidirectional, Read/remove/replace, only for list implementation.

09/11/17

Eg:
```
interface A
{
  void m1();
}
class X
{
  private class Y implements A
  {
```

```
public void m1()
{
  ==
}
```

```
X) A a = new Y();
private class
a.m1();
```

```
X x = new X();
A a = x.getInstance();
a.m1();
```

```
public A getInstance()
{
  return new Y();
}
```

Helper method

```
}
```

```
class AL implements Collection
{
  private class xyz implements Iterator
  {
    Public boolean hasNext()
    {
      ==
    }
    public Object next()
    {
      ==
    }
    public void remove()
    {
      ==
    }
  }

  public Iterator Iterator()
  {
    return new xyz();
  }
}.
```

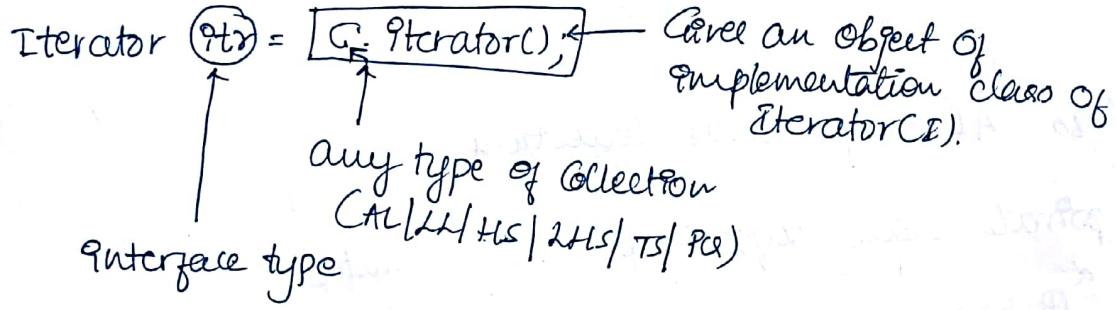
```
interface Iterator.
{
boolean hasNext();
Object next();
Void remove();
}.
```

- Helper methods.

i) public Iterator iterator() :-

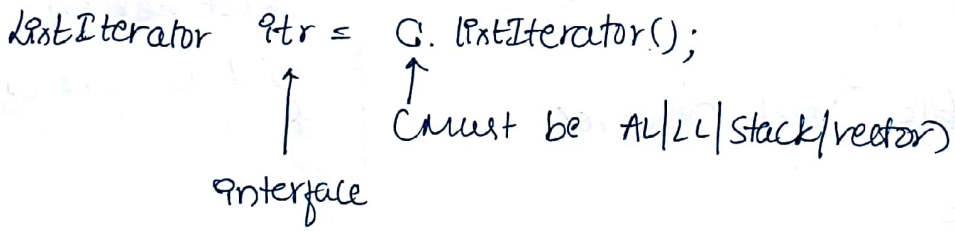
→ This helper method is used to get the implementation class object of Iterator interface.

→ We cannot create the implementation class object of iterator interface directly from outside This is because the class which implements iterator interface is a private inner class.



ii) Public ListIterator listIterator():

→ This helper method is used to obtain the implementation class object of ListIterator interface.



Program:

① write a program to display only string objects from hashset collection.

```
package jsp.jcf;
import java.util.HashSet;
import java.util.Iterator;
public class HashSetDemo
{
public static void main(String[] args)
{
  HashSet h= new HashSet();
  h.add(10);
  h.add("asp");
```

```
h.add ("Hello");
h.add (432.32);
Iterator ptr = h.iterator();
while (ptr.hasNext())
{
    Object obj = ptr.next();
    if (obj instanceof String)
    {
        S.O.P (obj); // [ @sp, Hello ]
    }
}
}
```

② Program to display string objects and remove integer objects from the hashset.

Soln:-

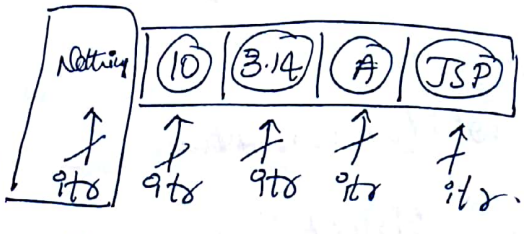
```
Iterator ptr = h.iterator();
while (itr.hasNext())
{
    Object obj = ptr.next();
    if (obj instanceof String)
    {
        S.O.P (obj);
    }
    else if (obj instanceof Integer)
    {
        ptr.remove();
    }
}
```

Assignment

③ Program to display all the students whose marks is greater 60 and less than 70. Remove all the students from the hashset whose marks is less than 60.

Working of ptr

```
Iterator ptr = al.iterator();
While (ptr.hasNext())
{
    ptr.remove(); // Illegal Exception.
}
```



(2)

* Linked Hashset

It is exactly same as hashset except the following
→ Insertion order is preserved & duplicates are not allowed

Differences

Hashset	Linked Hashset
→ underlined datastructure is Hashtable	→ unlined data structure is Hashtable + Doubly Linked list
→ Insertion order is not preserved.	→ Insertion order is preserved.

Eg:-

```
package jsp.jsf;
import java.util.Hashset;
public class LinkedHashSetDemo1
{
    P SV M (String[] args)
    {
        LinkedHashset h = new LinkedHashset();
        h.add (10);
        h.add ("jsp");
        h.add ("A");
        h.add (30);
        h.add (342.45);
        S.O.P(h); // [ 10, jsp, A, 30, ]
```

Assignment

```
package jsp.jsf;
import java.util.Hashset;
import java.util.Iterator;
Public class HashSetDemo2:
{
  P S V m (String[] args)
  {
    Hashset h= new Hashset();
      h.add (new Student ("Pooja", 65, "ECE"));
      h.add (new Student ("Peter", 75, "CSE"));
      h.add (new Student ("Nalme", 68, "ECE"));
      h.add (new Student ("Venus", 70, "CSE"));

  Iterator itr = h.Iterator();
    While (itr.hasNext())
    { object obj
      Student std = (Stude itr.next();
      S.O.P( std);
      if ( Std instance of Integer)
        { if( Student S = (Student) obj;
          if (S.marks >60 && S.Marks <70)
            {
              S.O.P (S);
            }
          else
            {
              h. itr.remove();
            }
        }
    } }
```

```
Package jsp.pj;
public class student
{
    string name;
    int marks;
    string branch;
    public student(string name, int marks, string branch)
    {
        this.name = name;
        this.marks = marks;
        this.branch = branch;
    }
    public string tostring()
    {
        return this.name "\t\t" + this.marks + "\t\t" + this.branch
    }
}
```

10/11/17

*Sorted set

→ Subinterface of set introduced in 1.2v.

→ If you want to represent a group of individual objects as a single entity. When the objects to be stored in some sorting order, then we will go for sorted set

→ Ascending order sorting is known as Default natural sorting order. Whereas Descending order sorting is known as Customised sorting

* Sorted set Methods :-

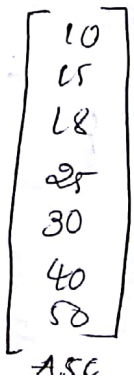
① first(); // 10

② last(); // 50

③ headset(); // [10,15, 18,25]

④ tailset(); // [18, 25, 30, 40, 50]

⑤ subset(15,40); // [15, 18, 25, 30]



* Tree Set :-

→ It is an implementation class of Navigable set introduced 1.2V.

→ The underlined data structure is Balanced Tree (Binary search Tree)

→ Duplicate objects are not allowed.

→ Insertion order is not preserved. objects will be inserted based on some sorting order.

→ Null insertion is not possible.

→ Heterogeneous objects are not allowed.

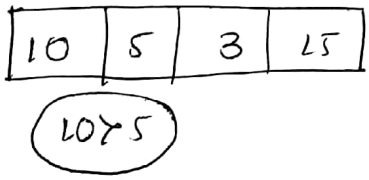
→ It implements serializable interface, and clonable interface but not random Access.

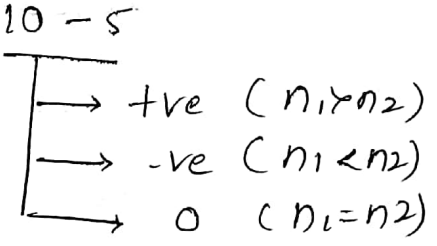
→ Treeset is a best choice if the objects to be stored in some sorting order.

→ Into the treeset we can add only homogeneous type of objects and also the objects that we are trying to add should be eligible for comparison.

→ An object is said to be eligible for comparison *if the objects are of same type and *the object should implement Comparable interface (or) comparato interface

→ Internally the objects treeset will be compared by subtracting, not by using relational operator.





→ By default objects of the treeset will be sorted in default natural sorting order. If we want in customised sorting order we should implement comparator (I).

* Constructors of Treeset :-

① Treeset t = new Treeset(); (DNSO)

② Treeset t = new Treeset (Comparator C); (CSO)

③ Treeset t = new Treeset(Collection C);

* Comparable (I) :-

→ ~~Interface~~ It is available in java.lang package.
→ It is meant for DNSO.
→ It has only one abstract method.

```
Interface Comparable.
{
  public int CompareTo (Object obj);
}
```

Obj1.compareTo (Object obj2)
↓ Current obj
↓ Given obj.

Obj1 - Obj2
- → +ve (obj1 > obj2)
- → -ve (obj1 < obj2)
- → 0 (obj1 = obj2)

Eg:

```
Package jsp.jsf;
import java.util.Treeset;
public class TreeSetDemo1
{
  P S V m (String[] args)
  {
    S.O.P ("**  ....  **");
    Treeset t = new Treeset();
    t.add (10);
    t.add (32);
    t.add (4);
    t.add (52);
```

Exception thrown:
// t.add ("ASP"); // Heterogeneous object not allowed
// t.add (Null); // Null not allowed

t.add(10); Duplicate Not allowed (doesnot display in o/p)
S.O.P(t); [4 10 32 52]
S.O.P("xx ..... xx");
}
}

logic for Character

```
t.add('A');
t.add('Z');
t.add('I');
t.add('G');
t.add('M');
```

o/p: [A G I M Z]

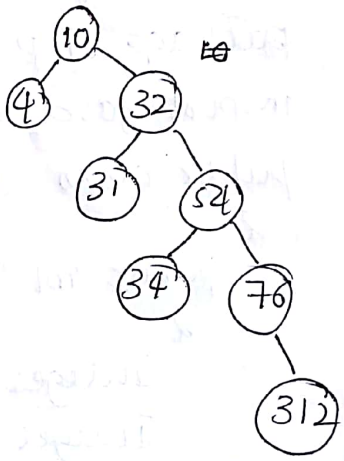
logic for String

```
t.add("Nalme");
t.add("Deeksha");
t.add("Pooja");
t.add("JSP");
t.add("QSP");
```

["Deeksha", "JSP", "Nalme", "Pooja", "QSP"].

Eg: t.add(10)
t.add(32)
t.add(4)
t.add(54)
t.add(34)

32. CompareTo(10)
32. CompareTo(54)
10. CompareTo(4)
32. CompareTo(54)
34. CompareTo(32)
34. CompareTo(10)
34. CompareTo(32)
34. CompareTo(54)



t.add(76)
76. CompareTo(10)
76. CompareTo(32)
76. CompareTo(54)

t.add(31)
31.
31.

t.add(312)
312.
312.
312.
312.

14/11/17

* Comparator Interface

→ It is present in java.util package
→ It is meant for customized sorting order.
→ It has the following 2 abstract methods.

(i) public int Compare (Object obj1, Object obj2);
current obj, given obj.

(ii) public boolean equals (Object obj);

* Differences.

Comparable	Comparator
→ present in java.lang	→ present in java.util
→ used for DNSO	→ used in CSO.
→ 1 method	→ 2 methods.

* Write a program to sort the treeset integer object in descending order.

```
package jsp.jcf;
import java.util.Comparator;
public class NumberDesc implements Comparator.
{
  public int compare(Object obj1, Object obj2)
  {
    Integer itr1 = (Integer) obj1; // downcasting
    Integer itr2 = (Integer) obj2; // downcasting.
    return itr2 - itr1; // auto unboxing.
```

```
package jsp.jcf;
public class TreeSetDemo24
{
   P S V m (String[] args)
   {
      S.O.P ("*** ..... ***");
      Treeset t = new Treeset (new NumberDesc());
       t.add(10);
       t.add(72);
       t.add(32);
```

```
t.add(42);
SOP(t);
SOP("**  .....  **");
}
}
```

* Write a program to sort the alphabets of Treeset in customised sorting order.

```
package jsp.jcf
import java.util.Comparator;
public class ~~Numbers~~ CharacterDesc implements Comparator
{
  public ~~class~~ int Compare(Object obj1, Object obj2)
  {
    Character ptr1 = (Character) obj1;
    Character ptr2 = (Character) obj2;
  { return ptr2 - ptr1;  // Integer will be returned
}
```

```
package jsp.jcf.
public class TreesetDemo
{
  PSVM(String[] args)
  {
    Treeset t = new Treeset(new CharacterDesc());
      t.add('N');
      t.add('P');
      t.add('S');
      S.O.P(t);
  }
}
```

* Program to sort the double objects in Treeset in descending order.

```
package jsp.jcf;
import java.util.Comparator;
public class NumberDesc implements Comparator
{
  public int Compare(Object obj2, Object obj2)
```

```
Double ptr1 = (Double) obj1;
Double ptr2 = (Double) obj2;
    return (int)ptr2-ptr1;
}
}
```

```
package jsp.jcf;
public Class TreesetDemo
{
  P S V m (String[] args)
  {
    Treeset t= new Treeset(new NumberDesc());
      t.add( 432.32);
      t.add ( 8.14);
      t.add ( 3.14);

      S.O.P (t);
  }
}
```

* write a program to sort the string objects of treeset in customised sorting order.

```
Class NumberDesc implements Comparable.
{
  public int Compare(object obj1, object obj2)
  {
    Double String s1 = (String) obj1;
           String s2 = (String) obj2;
           return s2.CompareTo (s1);
  }
}
main.
```

* program

```
package jsp.jcf.
Public class Student implements Comparable
{
  String name;
    int id;
    double marks;
```

```
public Student (String name, int id, double marks)
{
  this.name = name;
  this.id = id;
  this.marks = marks;
}
public String toString()
{
  return this.name + "\t\t" + this.id + "\t\t" + this.marks;
}
public int CompareTo(Object obj)
{
  Student s = (Student) obj
  return this.name.CompareTo(s.name); // names in ax
  // return this.id - s.id; // id in asc.
  // return (int) (this.marks - s.marks; // marks in asc
}
}
package jsp.jcf;
import java.util.Iterator;
import java.util.TreeSet;
public class TreeSetDemo
{
public static void main (String[] args)
{
  TreeSet t = new TreeSet();
  t.add ( new Student("Dinga", 10, 86.00));
  t.add ( new Student("Dingi", 20, 87.00));
  S.O.P ("Name\t\t Id \t\t Marks");
  S.O.P("  - — — - - - — - ");
  Iterator itr = t.iterator();
  while (itr.hasNext())
  {
    Student s = (Student) itr.next();
    S.O.P(s);
```

```
package jsp.jcf;
import java.util.comparator;
public class student implements Comparable, Comparator
{
  String name;
    int id;
  double marks;
public student (String name, int id, double marks)
  {
    this.name = name;
    this.id   =id;
    this.marks= marks;
  }
  public int compareTo( Object obj)
  {
    Student s = (Student) obj;
    return this.name.compareTo( s.name);
  }
  public int compare( Object obj1, Object obj2)
  {
    Student s1 = (Student) obj1;
    Student s2 = (Student) obj2;
     return s2.id= s1.id;
  }
}
package jsp.jcf;
import java.util.Treeset;
public class TreeSetObjects
{
  public void addObjects(Treeset t)
  {
    t.add( new Student ("Dinga", 10, 86.00));
    t.add ( new Student (" Dingi", 20, 81.00));
    S.O.P("Name \t\t Id \t\t Marks");
    S.O.P("  ---------- ");
    Iterator its = t.iterator();
```

```
while (itr.hasNext())
{
Student s = (Student) itr.next();
  s.o.p(s);
}
}
}

package jsp.jq;
import java.util.Scanner;
import java.util.Treeset;
public class TreesetDemo
{
  public static void main(String[] args)
  {
    for (;;)
    {
      s.o.p("1.DNLSO\n2.eSO\n3.exit");
      s.o.p("Enter your choice");
      Scanner sc= new Scanner (System.in)
      int ch = SO.nextInt()
      TreesetObjects ts = new TreesetObjects ();
      switch(ch)
      {
        case 1 : ts.addObjects (new Treeset ());
                 break;
        case 2 : ts.addobjects (new Treeset(new
                                Student(null,0,0.0)
                 break;
        case 3 : system.exit(0);
        default: s.o.p ("Invalid choice");
      }
}}}
```

* Program to sort treeset employee objects in descend
① order of the salary.

* Program to display all the students whose mark
② is greater than 60 and less than 70. o/p should
be in the increasing order of the marks.

* program to display all the movies in decreasing
③ order of the year of release.

* program to display all the Car objects in
④ increasing order of the price.

Soln

```
public class Employee implements Comparable
{
  String name;
    int id;
  double sal;
  Public Employee(String name, int id, double sal)
   {
     this.name = name;
     this.sal = sal;
     this.id = id;
   }
  public String toString()
 {
  return this.name"\t\t"+this.id+"\t\t"+this.sal;
 }
public int Compare(Object Obj1, Object Obj2)
 {
     Double d1 = (Double) Obj1;
     Double d2 = (Double) Obj2;

     return d2.CompareTo(d1);

 }
```

```
public class TreeSetDemo
{
    p s v m (String[] args)
    {
        TreeSet t = new TreeSet(new Compare
        t.add(new Employee("Ram", 100, 50000));
        t.add(new Employee("Shri", 400, 10,000));
        S.O.P("name \t\t id \t\t salary");
        Iterator itr = t.iterator();
        while(itr.hasNext())
        {
            Employee e = (Employee) itr.next();
            SOP(e);
        }
    }
}
```

② public class Student implements Comparable

```
{
    String name;
    int id;
    double marks;
    public Student(String name, int id, double marks)
    {
        this.name = name;
        this.id   = id;
        this.marks = marks;
    }
    public String toString()
    {
        return this.name+"\t\t"+this.id+"\t\t"+this.mar
    }
    public int compareTo(Object obj)
    {
        Student s = (Student) obj;
        return this.marks.compareTo(s.marks);
    }
```

```
public class Treeset
{
  public void main(String[] args)
  {
    Treeset t= new Treeset();
    t.add( new student("Dinga", 10, 86.00));
    t.add (new student(" Roopa,"30, 95.00));
    SOP (" Name \t\t id \t\t marks");
     Iterator itr= t.iterator();
    while( itr. hasNext())
    {
      Object obj = itr.next();
      if (obj instanceof student)
      {
        Student s' = (student) obj;
        if (s.marks > 60 && s. marks< 70)
        {
          S.O.P (obj);
        }
      }
    }
}
```

③ public class movie implements comparator

```
{
  String name;
  String type ;
    int year ;
  public movie(String name, String type, int year)
  { this. name = name;
    this . type = type;
    this . year = years
  }
  public String toString()
  {
  return this.name+"\t\t"+this.type+"\t\t"+this.year;
```

```
public int compare(Object obj1, Object obj2)
{
    Integer i1 = (Integer) obj1;
    Integer i2 = (Integer) obj2;
    return i2 - i1;
}
public class TreeSetDemo
{
    p.s.v.m(String[] args)
    {
        TreeSet t = new TreeSet(new movie());
        t.add(new movie("Twilight", "Drama", 2004));
        t.add(new movie("DDLJ", "Romance", 1995));
        t.add(new movie("YJHD", "Romance", 2013));
        S.O.P("name \t\t type \t\t year");
        Iterator itr = t.iterator();
        while(itr.hasNext())
        {
            movie m = (movie) itr.next();
            S.O.P(m);
        }
    }
}
```

(4)
```
public class Car implements Comparable
{
    String name;
    String color;
    double price;
}
public Car(String name, String color, double price)
{
    this.name = name;
    this.color = color;
    this.price = price;
```

```
public String toString()
{
return this.name+"\t\t"+this.color+"\t\t"+this.price;
}
public int compareTo(Object obj)
{
  Car c = (Car) obj;
  return (int) (this.price - c.price);
}
public class TreeSetDemo
{
  P S V M (String[] args)
  {
    Treeset t = new Treeset
    t.add(new Car("Figo","White",100000);
    t.add(new Car("Zen","Red",10,5000);
    t.add(new Car("910","Black",80,0000);
    S.O.P("Name\t\t Color\t\t Price");
    Iterator ptr = t.iterator();
    while(itr.hasNext());
    {Car c = (Car) ptr.next();
      S.O.P(c);
    }
}
```

12/11/17

* Queue Interface :-

Sub interface of collection introduced 1.5V. If we want to represent a group of individual objects where the objects needs to be processed based on FIFO then we will go for queue.

* Queue specific methods. (To be used only in priority Queue)

① public void offer(object obj) :- To add an obj into P.Q

② public Object poll();
→ process head queue element (consd), if Queue has element. After processing remove elements from Q
→ If Q is empty, it returns null.

③ public object remove();
→ process head element (consd), if Q has elements. After processing remove the element from the Q
→ If Q is empty, it throws exception.

④ public object peak();
→ process head element (consd), if queue has elements. After processing it will not remove from the queue
→ If Q is empty, it returns null.

⑤ public object element();
→ process head element (consd), if queue has elements After processing it will not remove from queue
→ If Q is empty, it throws exception.

* Priority Queue

→ It is an implementation class of Queue introduced in 1.5V

→ The underlined data structure is priority queue.

→ Duplicates are allowed.

→ Insertion order is not preserved. The objects will be inserted based on some priority.

→ Heterogenous objects are not allowed.

→ ~~It~~ Null insertion is not possible.

→ It implements Serializable interface, Clonable(I) but not Random Access.

Note: In priority queue objects will be inserted based on the priority and processing will be done either based on default natural sorting order or CSO.

* Constructors of Priority Queue:

① PriorityQueue pq = new PriorityQueue();
→ Creates an empty priority queue object with the default initial capacity level 11
→ DNSO.

② PriorityQueue pq = new PriorityQueue(int initialCapacity, Comparator c);
→ Creates an empty P-Q object with the specified initial capacity and specified Comparator type.

③ PriorityQueue pq = new PriorityQueue(Collection c);
→ Converts the given collection into priority queue equivalent.

Eg: ① Package jsp_34;
import java.util.PriorityQueue;
public class PriorityQueueDemo
{
p s v m(String[] args)
{

```
PriorityQueue pq = new PriorityQueue();
pq.add(10);
pq.add(23);
pq.add(3);
pq.add(52);
S.O.P("Before processing = " + pq.size());
Object obj = pq.poll();
while(obj != null)
{
  Integer ptr = (Integer) obj;
  S.O.P(ptr);
  obj = pq.poll();
}
S.O.P("After processing size = " + pq.size());
}
```

② Process the P.Q objects in customised sorting order

```
package jsp-jcf
import java.util.PriorityQueue
public class PriorityQueueDemo
{
  P S V M ()
  {
    PriorityQueue pq = new PriorityQueue(10, NumberDe
    pq.add(10);
    pq.add(32);
    pq.add(5);
    S.O.P(pq);
  }
}

public class NumberDesc implements Comparator
{
  public int compare(Object obj1, Object obj2)
  {
    Integer ptr1 = (Integer) obj1;
    Integer ptr2 = (Integer) obj2;
    return ptr2 - ptr1;
```