21/02/11

Language fundamentals

1) Identifier :-

→ A name in Java program is called identifier, it can be class name or variable name or method name or label name.

Ex1:

```
class Test  → classname
     ✓
{                → method name
p.s.v. main(String [] args)
        ✓     ✓        ✓
{
   int x=10;                  ✓→ is identifiers.
       ✓
       ↳ variable name
}
}
```

* Rules to define identifiers :-

1) The only allowed characters in Java identifier are :

✓ (a to z
A to Z
0 to 9
_
$)

→ If we are using any other character we will get Compiletime Error

Ex :-

✓ all_member

✗ all#

✓ _$_$

✗ 098$_10

2) identifier Can't starts with digit. Ex:- ✗ 123total

✓ total123.

3). Java identifiers are CaseSensitive.

```
class Test
{
 int Number =10;
 int NUMBER =20;
 int NumBer = 30;
}
```

} we Can differenciate w.r.t Case.

4) there is no Length Limit for Java identifiers. but it's not recommended to take morethan 15 length (>15).

5) Reserved words Can't be used as identifiers.

5) All predefined Java class names & interface names we Can use as identifiers. ~~but~~ Eventhough it is legal, but it is not recommended.

Ex:-

✓
```
Class Test
{
 int String=10;
 S.o.pln(String); 10
}
```

✓
```
Class Test
{
 int Runnable = 20;
 S.o.pln(Runnable); 20
}
```

Q) which ~~are~~ the following are valid Java identifiers?

✓ ① Java2Share

✗ ② 4shared

✗ ③ all@hands

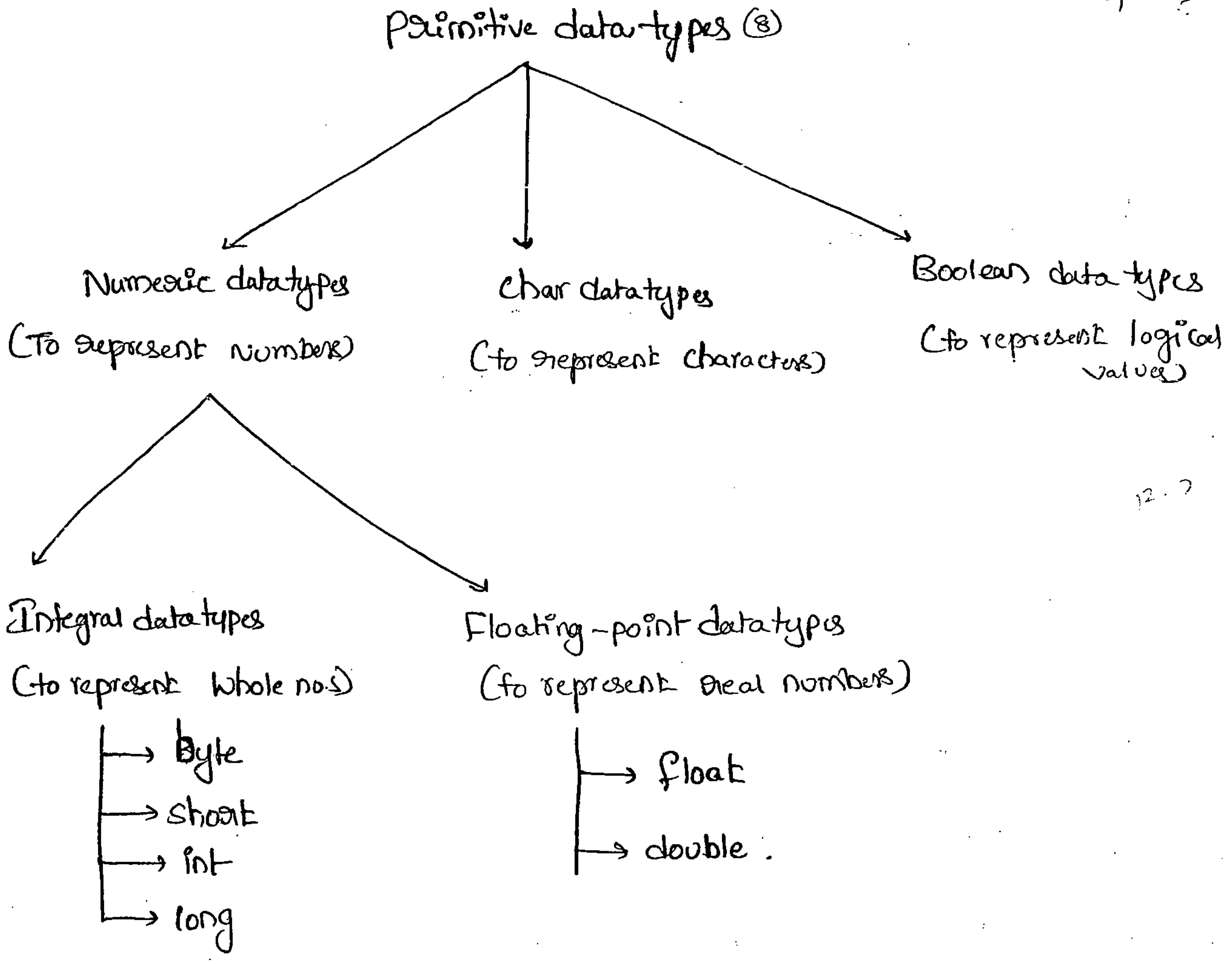
✓ ④ total_no_of_Students

✓ ⑤ _$_

✗ ⑥ total#

✗ ⑦ int

✓ ⑧ Integer



Primitive data-types (8)

Numeric datatypes (To represent numbers)

Char datatypes (to represent characters)

Boolean data types (to represent logical values)

Integral datatypes (to represent whole nos)

- byte
- short
- int
- long

Floating-point datatypes (to represent real numbers)

- float
- double.

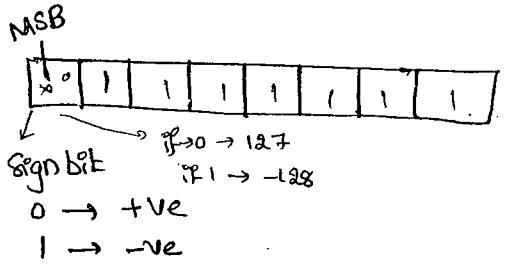
① Byte :-

Size = 8-bits (or) 1 Byte

Max_value = 127

Min_value = -128

Range = -128 to +127



→ The Most Significant Bit is Called "Sign bit". 0 means +ve value, 1 means -ve value.

→ +ve numbers represented directly in the memory where as -ve numbers represented in 2's Complement form.

Ex:-

✓ byte b = 100;

✓ byte b = 127;

✗ byte b = 130; C.E!- possible loss of precision
found : int
Required : byte

✗ byte b = 123.456; C.E!- PLP
found : double
Required : byte

✗ byte b = true ; C.E!- ~~PLP~~ incompatable types
found : boolean
Required : byte

✗ byte b = "durga"; C.E:- incompatible types
found : ~~java~~ java.lang.String
required : byte.

→ byte datatype is best suitable if we want to handle data in terms of streams either from the file or from the Network.

② Short :-

Size : 2-bytes (16-bits)

Range : -2^{15} to $2^{15}-1$

[-32768 to 32767]

Eg!- ✓ Short s = 32767

✓ Short s = -32768

✗ Short s = 32768 C.E!- PLP
found : int
required : short

X Short s = 123.456   C.E:- PLP

found : double

Required : Short

X Short s = true   C.E:- Incompatible types

found : boolean

Required : Short .

→ Most rarelly used datatype in Java is "Short"

→ Short datatype is best suitable if we are using 16-bit processors like 8086 but these processors are Completly out dated & hence Corresponding Short data-type is also out dated .

③ int :-

→ The most Commonly used datatype is "int"

Size : 4-bytes

Range : -2^{31} to $2^{31}-1$

[-2147483648 to 2147483647)

Note:-

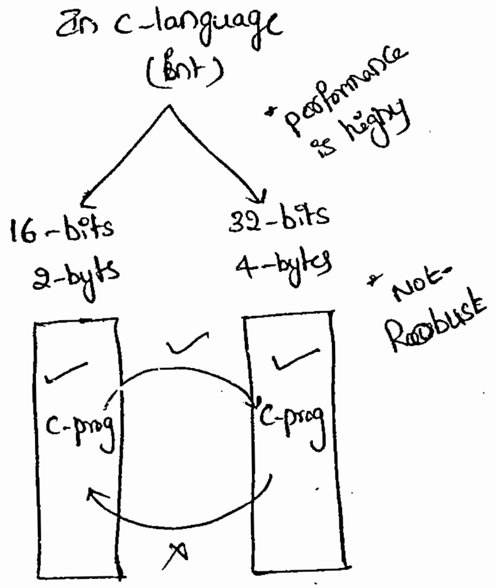
→ In C language the size of int is varied from platform to platform for 16-bit processors it is 2-bytes but for 32-bit processors it is 4-bytes

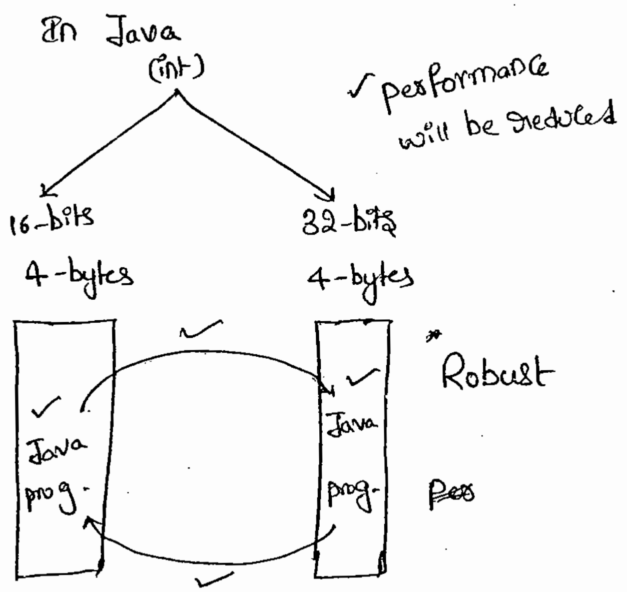
* The main advantage of this approach is read & write operation we can perform Very effiently and performance will be improved. But the main

* disadvantage of this approach is the chance of ~~[illegible]~~ failing C program is very very high if we are changing platform. Hence C-language is not Considered as Robust .

→ But in Java the Size of int is always 4-bytes irrespective of any platform. * The main advantage of this approach is the chance of failling Java program is very very less, If we are changing underlaying platform & Hence Java is Considered as Robust langu

* But the main disadvantage in this approach is read & write operations will become Costly & performance will be reduced.





13/02/11

4) long :-

→ when ever int is not enough to hold big values then we should go for long data type.

Ex(1):- To represent the amount of distance travelled by light in 1000 days int is not enough Compulsary we should go for long type

Ex:-1) long l = 1,23,000×60×60×24×1000 miles;

2)

Ex(2):-

To Count the no. of characters present in a big file. int may not enough Compulsary we should go for long data type.

Size = 8 bytes

Range = -2^{63} to $2^{63}-1$

Note:-

→ All the above data-types (byte, short, int, long) ment for representing whole values.

→ If we want to represent real numbers Compulsary we should go for floating point data types.

Floating Point data-types :-

floating point data-types

float	double
1) Size : 4-bytes	1) Size : 8-bytes
2) Range : -3.4e38 to 3.4e38	2) Range : -1.7e308 to 1.7e308
3) If we want 5 to 6 decimal places of accuracy then we should go for float	3) If we want 14 to 15 decimal places of accuracy then we should go for double.
4) float fallows single precision	4) double fallows double precision

Boolean data type :-

Size : Not Applicable (Virtual machine dependent)

Range : Not Applicable [But allowed values are true/false]

Q) which of the following boolean declarations are valid

X 1) `boolean b = 0;` C.E:- incompataible types
found : int
required : boolean

✓ 2) `boolean b= true;`

X 3) `boolean b = True;` C.E:- Can't find symbol
Symbol : Variable True
Location : class Test

X 4) `boolean b= "false"` C.E:- incompatible types
found : java.lang.String
required : boolean

✓ 5)

```
boolean True = true
boolean b = True
S.o.pln(b);   true
```

Ex:-

```
int x=0;
if(x)
{
S.o.pln("Hello");
}
else
{
S.o.pln("Hi");
}
```

in Java X

C++ ✓

```
while(1)
{
  S.o.pln("Hello");
}
```

in Java X

in C++ ✓

C.E:- incompatible types
found : int
required : boolean

→ The only allowed values for the boolean datatypes are "true" or "false" where case is important.

Char datatype :-

→ In ~~old~~ languages like C & C++ we can use only ASCII characters and to represent all ASCII characters 8-bits are enough. hence char size is 1-byte.

→ But in java we can use unicode characters which covers world wide all alphabetes sets. The no. of unicode characters is ">256" & hence 1-byte is not enough to represent all characters compulsary we should go for 2-bytes.

Size : 2-bytes

Range : 0 to 65535

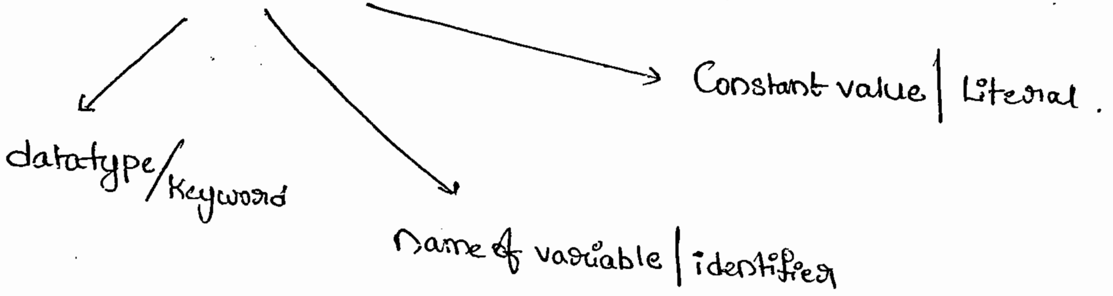
Summary of primitive data types :-

datatype	Size	Range	Corresponding Wrapper classes	default value
byte	1-byte	-2^7 to 2^7-1 [-128 to 127]	Byte	0
Short	2-bytes	-2^{15} to $2^{15}-1$ [-32768 to 32767]	Short	0
int	4-bytes	-2^{31} to $2^{31}-1$ [-2147483648 to 2147483647]	Integer	0
long	8-bytes	-2^{63} to $2^{63}-1$	Long	0
float	4-bytes	-3.4e38 to 3.4e38	Float	0.0
double	8-bytes	-1.7e308 to 1.7e308	Double	0.0
Char	2-bytes	0 to 65535	Character	0 [represents blank space]
boolean	NA	NA [true/false are allowed]	Boolean	false (True in C++)

Literals :-

↳ A Constant value which Can be assign to the Variable is Called "Literal"

Ex:- int x = 10 ;



Integral Literals :-

→ for the Integral datatypes (byte, short, int, long) the following are Various ways to Specify Literal value

1) decimal literals:-

allowed digits are 0 to 9

Ex:- int x = 10;

2) Octal literals:-

→ allowed digits are 0 to 7

→ literal value should be prefixed with "0" [zero]

Ex:- int x = 010;

3) Hexadecimal literals:

→ allowed digits are 0 to 9, a to f or A to F

→ for the Extra digits we Can use both upper case & lower Case. This is one of very few places where Java is not case sensitive

→ Literal value should be prefixed with 0x (or) 0X

Ex:- int x = 0x10
(or)
int x = 0X10

→ These are the only possible ways to specify integral literal.

Ex:-
```
class Test
{
 p.s.v.m(String [] args)
 {
  int x = 10;
  int y = 010;
  int z = 0X10;
  S.o.pln(x+"----"+y+"------"+z);
 }
}
          10        8         16
```

$(10)_8 = (?)_{10}$

$0\times8^0 + 1\times8^1 = 8$

$(10)_{16} = (?)_{10}$

$0\times16^0 + 1\times16^1 = 16$

21/02/11

Q) which of the following declarations are valid.

✓ ① int x = 10;

✓ ② int x = 066;

✗ ③ int x = 0786; C.E:- integer number too large

✓ ④ int x = 0XFACE; 64206

✗ ⑤ int x = 0XBEER; C.E:- (after Be) ; Expected

✓ ⑥ int x = 0XBea; 3050

→ Bydefault Every integral literal is of int type but we Can Specify Explicitly as long type by Suffixing with l or L.

Ex:-

✓ 1) int i = 10;

X 2) int i = 10l;   C.E:- PLP
found : long
required : int

✓ 3) long l = 10l;

✓ 4) long l = 10;

→ There is no way to Specify integral literal is of byte & short types Explicitly.

→ If we are assigning integral literal to the byte variable & that integral literal is with in the range of byte then it treats as byte literal automatically. Similarly short literal also.

Ex:- byte b = 10; ✓

byte b = 130; X   C.E:- PLP
found : int
Required : byte

Floating point Literals:-

→ Every floating point literal is bydefault double type & hence we Can't assign directly to float variable.

→ But we Can Specify Explicitly floating point literal is the float type by Suffixing with 'f' or 'F'.

Ex:- X float f = 123.456;   C.E: P.L.P
found : double
required : float

✓ float f = 123.456f;

✓ double d = 123.456;

→ We Can Specify floating point literal Explicitly as double type by Suffixing with d or D .

Eg:- ✓ double d = 123.4567D;

X float f = 123.4567d; C.E:- PLP

found : double
required : float

→ We Can Specify floating point literal only in decimal form & we Can't Specify in octal & Hexa decimal form.

Ex:-

✓ 1) double d = 123.456;

✓ 2) double d = 0123.456; o/p:- 123.456

X 3) double d = 0x123.456; C.E:- malformed floating point literal

Q) which of the following floating point declarations are Valid?

X 1) float f = 123.456;

✓ 2) double d = 0123.456;

X 3) double d = 0x123.456;

✓ 4) double d = 0xface; //64206.0

✓ 5) float f = 0xBea;

✓ ⑥ float f = 0642; //418.0

Because these 3 are not floating point So, that values are taking int type.

→ We Can assign integral literal directly to the floating point datatypes & that integral Literal Can be Specified either in decimal form or octal form or hexa decimal form.

double
→ But we Can't assign floating point literals directly to the integral types.

Ex:- ✗ int i = 123.456;   PLP
found : double
required : int

✓ double d = 1.2e3;
S.o.pln(d) ; 1200.0

→ we Can Specify floating point Literal even in Scientific form also [exponetial form]

ex:- ✓ 1) double d = 1.2e3;
S.o.pln(d) ; 1200.0

✗ 2) float f = 1.2e3;   C.E:- PLP
found : double
required : float

✓ 3) float f = 1.2e3f;
o/p:- 1200.0

Boolean Literals:-

→ The only possibles values for the Boolean data types are true/false

Q) which of the following Boolean declarations are valid.

✗ ① boolean b = 0;   C.E:- Incompatible types
found : int
required: boolean

✗ ② boolean b = True;   C.E:- Can't find Symbol
Symbol : variable True

✓ ③ boolen b = true;

✗ ④ boolen b = "true";   C.E:- Incompatible types
found : java.lang.String   required : boolean.

1) Ex:-

```
int x=0;
if(x)
{
S.o.pln("Hello");
}
else
{
S.o.pln("Hi");
}
```

```
while(1)
{
S.o.pln("Hello");
}
```

C.E:- incompatible types
found : int
required : boolean

Ex②:-

✗

```
int x =10;
if(x = 20)
{
S.o.pln("Hello");
}
else
{
S.o.pln("Hi");
}
```

C.E:- IT
f : int
R : boolean

✓

```
int x=10;
if (x == 20)
{
S.o.pln("Hello");
}
else
{
S.o.pln("Hi");
}
```

O/P: Hi

✓

```
boolean b =true;
if(b = false)
{
S.o.pln("Hello");
}
else
{
S.o.pln("Hi");
}
```

O/P:- Hi

```
boolean b =true;
if(b== true)
{
S.o.pln("Hello");
}
else
{
S.o.pln("Hi");
}
```

O/P:- Hello.

Char Literals :-

1) A char literal can be represented as single character with in single codes

Ex:- ✓ char ch = 'a';

✗ char ch = a; C.E:- Can't find Symbol
Symbol : variable a
location : class xxxx

✗ char ch='ab';
→ C.E:- unclosed character literal
C.E : unclosed " "
C.E : not a statement

25/08/11:-

2) → A char literal can be represented as integral literal which represents unicode of that character.

→ we can specify integral literal either in decimal form or octal form or Hexa decimal form. But allowed range 0 to 65535.

Ex:- ✓ 1) char ch = 97;
S.o.p(ch); a

✓ 2) char ch = 65535;
S.o.pln(ch);

✗ 3) char ch = 65536; C.E:- PLP
found : int
required : char

✓ 4) char ch = OXFACE;

✓ 5) char ch = 0642;

3→ A char literal Can be represented in Unicode representation which is nothing but $\boxed{\backslash u\ \underline{xxxx}}$ 4-digit hexa decimal no.

Ex:- ✓ 1) char ch = '\u0061';

S.o.p(ch); a

✗ 2) char ch = '\uabcd; → semicolon missing

✓ 3) char ch = '\uface';

✗ 4) char ch = '\i beaf';

4→ Every escape character is a char literal

Ex:- ✓ 1) char ch = '\n';

✓ 2) char ch = '\t';

✗ 3) char ch = '\l';

escape Character	meaning
\n	new Line
\t	horizontal tab
\r	Carriage Return
\b	Back space
\f	form feed
\'	Single quads
\"	Double quads
\\	Back slash

Q) which of the following are valid char declarations.

✓ 1) char ch = 0xbeaf;

✗ 2) Char ch = \u beaf ;   because ' '

✗ 3) char ch = -10 ;

✗ 4) char ch = '\x';

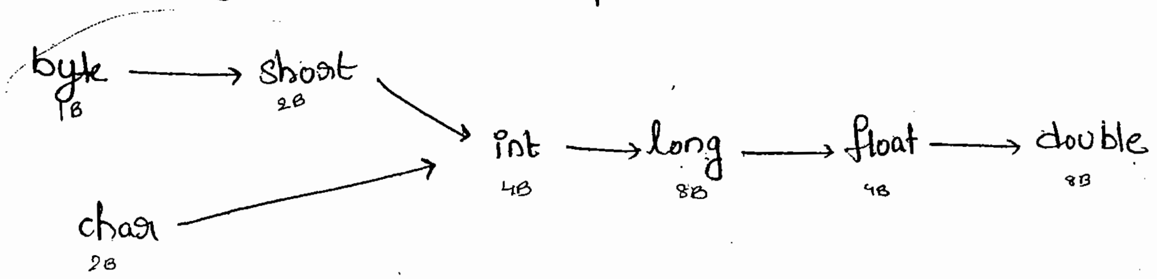
✓ 5) char ch = 'a';

String Literals :-

→ Any Sequence of characters with in "_" (double codes) is Called String literal.

Ex!- String s = "Java";

→ The following promotions will be performed automatically by the Compiler



Arrays

(20)

1. Array declaration
2. Array Creation.
3. Array Initialization.
4. Declaration, Creation, Initialization in a Single Line.
5. length vs Length()
6. Annonymous Array
7. Array element assignments
8. Array Variable assignments.

Array:-

→ An Array is an Indexed Collection of fixed no. of homogeneous data elements.

→ The main advantage of array is we Can represent multiple values under the Same name. So, that readability of the Code improved.

→ But the main limitation of array is Once we Created an array there is no chance of increasing/decreasing Size based on our requirement. Hence memory point of view arrays Concept is not recommended to use.

→ we Can resolve this problem by using Collections.

1) **Array declarations:-**

(a) **Single dimensional Array declaration:-**

✓ 1) `int[] a;`

✓ 2) `int a[];`

✓ 3) `int []a;`

→ 1st one recommended because Type is ~~Cl~~ Clearly Seperated from the ~~N~~ Name.

→ At the time of declaration we Can't Specify the Size.

Ex:- ✗ 1) `int[6] a;`

(b) **2D Array declaration:-**

✓ 1) `int[][] a;`

✓ 2) `int [][]a;`

✓ 3) `int a[][];`

✓ 4) `int[] a[];`

✓ 5) `int[] []a;`

✓ 6) `int []a[];`

c) 3D - Array declarations:-

1) int[][][] a;

2) int a[][][];

3) int [][][]a;

4) int[] [][]a;

5) int[] a[][];

6) int[] []a[];

7) int[][] []a;

8) int[][] a[];

9) int [][]a[];

10) int []a[][];

Q) Which of the following are valid declarations.

✓ 1) int[] a,b; a→1 b→1

✓ 2) int[] a[],b; a→2 b→1

✓ 3) int[] []a,b; a→2 b→2

✓ 4) int[] []a,b[]; a→2 b→3

✗ 5) int[] []a,[]b; a→2 b→3 C.E:-

→ If we want to specify the dimension before the variable it is possible only for the first variable.

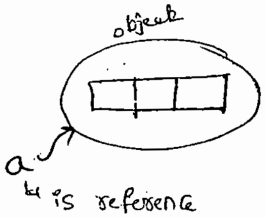
Ex:- int[] []a, []b;

Allowed not allowed;

2) Array Construction:-

→ Every array in Java is an object. hence we can create by using new operator.

ex:- int[] a = new int[3];



→ for Every array type Corresponding Classes are available but These classes are not applicable for programmer level.

Array type	Corresponding classname
① int[]	[I@ - - - - -
② int[][]	[[I@ - - - -
③ double[]	[D@ - - -
⋮	⋮

→ At the time of Construction Compulsary we should Specify the Size otherwise we will get C.E.

```
Ex!- int[] a = new int[];  ✗  C.E!-
     int[] a = new int[3]; ✓
```

→ It is legal to have an array with size 0 in java.

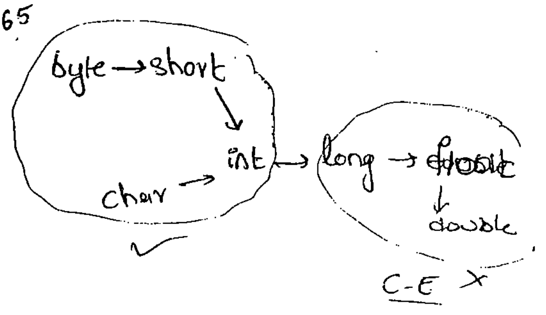
```
ex!- int[] a = new int[0]; ✓
```

→ If we are Specifying array Size as -ve int value, we will get runtime Exception Saying ~~→~~ NegativeArraySizeException.

```
Ex!- int[] a = new int[-6];  R.E!- NegativeArraySizeException
```

→ To Specify array Size The allowed data-types are byte, short, int char, If we are using any other type we will get C.E.

ex: ① ✓ `int[] a= new int['a'];` a=97 A=65



```
② byte b=10;
✓ int[] a=new int[b];
③ short s=20;
✓ int[] a = new int[s];
✗ int[] a= new int[10l];
✗ int[] a= new int[10.5];
```

Note:-

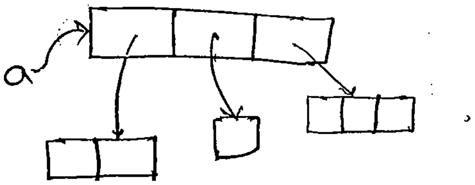
→ The max. allowed arraysize in java is 2147483647 (max. value of int datatype).

Creation of 2D-Arrays:-

→ In java multi dimensional arrays are not implemented in matrix form. They implemented by using Array of Array Concept.

→ The main advantage of this approach is memory utilization will be improved.

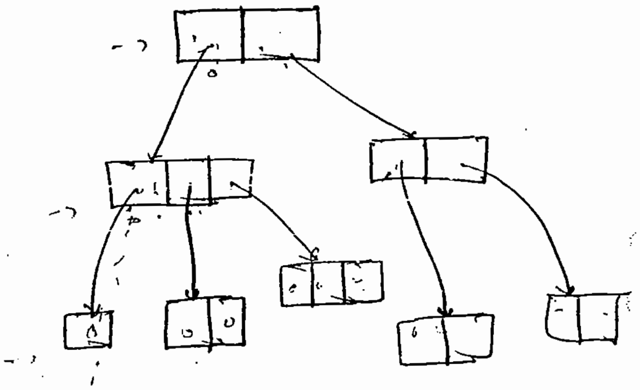
ex:-
```
int[][] a = new int[3][];
    a[0] = new int[2];
    a[1] = new int[1];
    a[2] = new int[3];
```



Note:-
In C++, ar

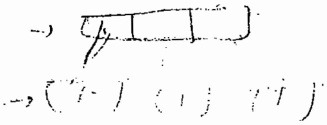
Ex 2:

```
int[][][]   a = new int[2][][];

a[0] = new int[3][];
a[0][0] = new int[1];
a[0][1] = new int[2];
a[0][2] = new int[3];
a[1] = new int[2][2];
```



Q:- which of the following Array declarations are valid?

```
X ① int[] a = new int[];
✓ ② int[][] a = new int[3][2];
✓ ③ int[][] a = new int[3][];
X ④ int[][] a = new int[][2];
✓ ⑤ int[][][] a = new int[3][4][5];
✓ ⑥ int[][][] a = new int[3][4][];
X ⑦ int[][][] a = new int[3][][5];
```



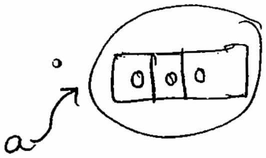
[2] this part = [ ]2

Array Initialization:-

→ Whenever we are Creating an array automatically every element is initialized with default values.

Ex(1):

```
int[] a = new int[3];
S.o.pln(a);      [I@3e25a5
                    └ hashcode
S.o.pln(a[0]);   0
```

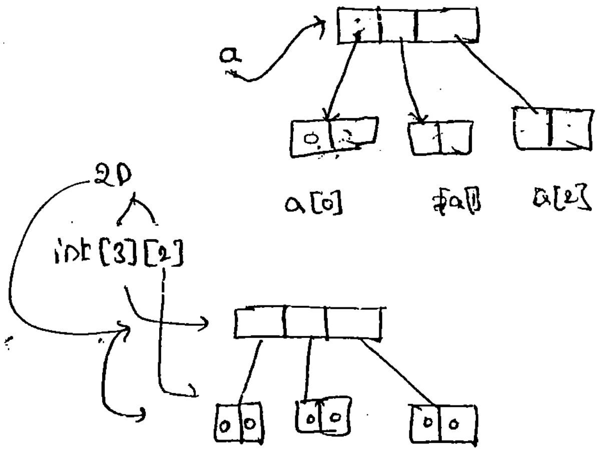


Note: Whenever we are trying to print any object reference internally toString() will be call which is implemented as follows.

Classname @ hexadecimal_String-of-hashCode.

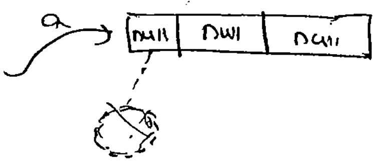
Ex(2):-

```
int[][] a = new int[3][2];

S.o.pln(a);         [[I@-----
S.o.pln(a[0]);      [I@4567
S.o.pln(a[0][0]);   0
```



Ex(3):-

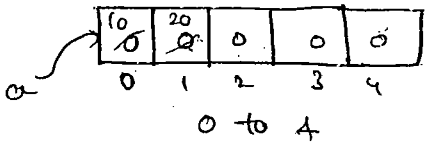
```
int[][] a = new int[3][];

S.o.pln(a);         [[I@-----
S.o.pln(a[0]);      null
S.o.pln(a[0][0]);   R.E: NPE
```



→ Once we created an array Every element by default initialized with default values. If we are not satisfy with those default values then we can override, those with our Customized values.

Ex:-

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

a[0] = 10;
a[1] = 20;
a[3] = 40;
a[50] = 50;    ——→ R.E: AIOBE
a[-50] = 60;   ——→ R.E: AIOBE
a[10.5] = 30;
     └——→ C.E:- PLP, found = double, required = int
```



Note:.

→ If we are trying to access an array with out of range index we will get RuntimeException Saying "AIOBE".

Array declaration, Construction & Initialization in a Single Line:-

→ We Can declare, Construct & Initialize an array into a Singleline.

Ex(1):-

```
int[] a;
a = new int[3];
a[0] = 10;
a[1] = 20;
a[2] = 30;
a[3] = 40;
```

⟹ `int[] a = {10, 20, 30, 40};`

char

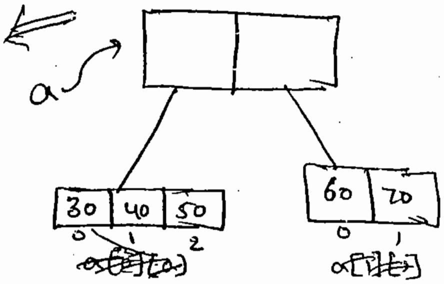
Ex(2):-

```
char[] ch = {'a', 'e', 'i', 'o', 'u'};
String[] s = {"Srinu", "Ravi", "Laxmi", "Sundar"};
```

→ We Can extend this shortcut Even for multidimensional arrays also.

Ex(3):-

```
int[][] a = {{30,40,50}, {60,70}};
```



→ We Can Extend this ShortCut Even for 3D array also

Ex:-

```
int[][][] a = {{{10,20,30}, {40,50}, {60}}, {{70,80}, {90,100}, {110}}};
```

Eg: int[][][] a = {{{10,20,30},{40,50},{60}},{{70,80},{90,100},{110}}}

S.o.pln(a[1][2][3]); RE:- AIOBE

S.o.pln(a[0][1][0]); 40

S.o.pln(a[1][1][0]); 90

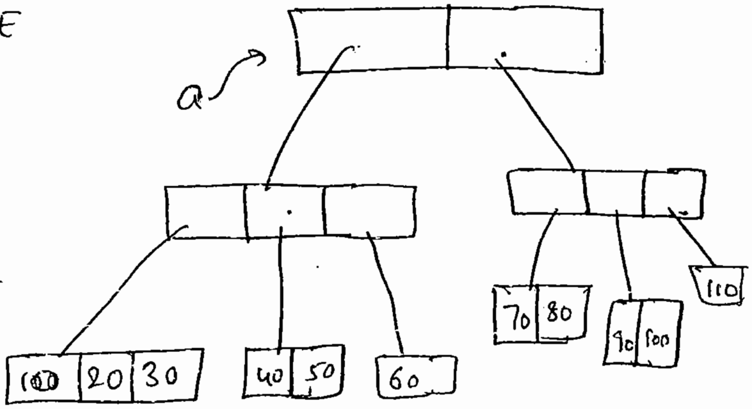
S.o.pln(a[3][1][2]); RE:. AIOBE

S.o.pln(a[2][2][2]); RE: AIOBE

S.o.pln(a[1][1][1]); 100

S.o.pln(a[0][0][1]); 20

S.o.pln(a[1][0][2]); R.E:- AIOBE



→ If we want to use Shortcut Compulsary we should perform declaration, Construction & initialization in a SingleLine.

→ If we are using multiple Lines we will get Compile-time Error.

Ex:-

int x=10; :-

✓ int x;

✓ x=10

int[] x = { 10, 20, 30 }:-

✓ int[] x ;

X x = {10, 20, 30};

C.E:- Illegal Start of Expression.

length() vs length :-

length :-

→ It is a final variable applicable only for arrays.

→ It represents the size of array

Eg:-
```
int[] a = new int[10];
    S.o.pln(a.length);   10
    S.o.pln(a.length()); C.E
```
Cannot find Symbol
Symbol : method length
location : class int[]

length() :-

→ It is a final method applicable only for String objects

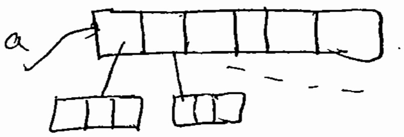
→ It represents the no. of characters present in String.

eg:-
```
String s = "durga";
   S.o.pln(s.length());  5
   S.o.pln(s.length);
```
C.E:- Cannot find Symbol
Symbol : variable length
location : java.lang.String.

→ In multidimensional arrays length variable represents only base size, but not total size.

Eg!- int[][] a = new int[6][3];

```
S.o.pln(a.length); 6
S.o.pln(a[0].length); 3
```



Note:-

→ length variable is applicable only for arrays where as length() is applicable for String objects.

Annonymous Array :-

→ Sometimes we Can Create an array with out name also Suchtype of nameless arrays are Called "Annonymus arrays".

→ The main objective of annonymous Array is Just for instant use. (not future). (only onetime)

→ We Can Create Annonymous Array as fallows.

```
new int[]{10,20,30,40}
```
✓

→ At the time of Annonymous Array Creation we Can't Specify the Size. Otherwise we will get Compiletime Error.

Eg!- X new int[4]{10,20,30,40}.

Eg:-

```
Class Test
{
P.S.v.main(String[] args)
{
```

```
Sum(new int[]{10,20,30,40});
}
public static void Sum(int[] x)
{
  int total=0;
 for(int x1 : x)
  {
   total = total+x1;
  }
 S.o.pln("the Sum: "+total);  100
}
```

↳ Based on our requirement we can give the name for Annoymous array, then it is <u>no longer Annonymous</u>.

Eg:-

```
String[] s = new String[]{"A","B"};
  S.o.pln(s[0]); A
  S.o.pln(s[1]); B
  S.o.pln(s.length); 2.
```

Array element assignments :-

Case(1) :-

→ for the primitive type arrays as Array elements we Can provide any type which Can be promoted to declare type.

① Eg:- ↳ for the int type arrays, the allowed Element types are byte, short, char, int. if we are Providing any other type ; We will get CompiletimeError.

Eg(1) :-

```
int[] a = new int[10];
✓ a[0] = 10;
✓ a[1] = 'a';
  byte b = 10;
✓ a[2] = b;
  short s = 20;
✓ a[3] = s;
✗ a[4] = 10l;   C.E:- PLP
                      found : long
                      required : int
✗ a[5] = 10.5;  C.E:- PLP, found : double
                           required : int
```

Eg(2):- for the float type array, the allowed Element types are byte, short, char, int, long, float.

$$\begin{tikzcd}
byte \arrow[r] & short \arrow[dr] & & & & \\
& & int \arrow[r] & long \arrow[r] & float \arrow[r] & double \\
& char \arrow[ur] & & & &
\end{tikzcd}$$

Case(2):-

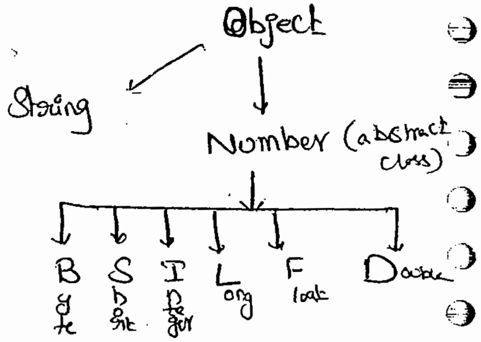
→ In The Case of Object type arrays as array elements we Can provide either declared type or its child class Objects.

eg1:- ① Number[] n = new Number[10];

```
✓ n[0] = new Integer(10);
✓ n[1] = new Double(10.5);
✗ n[2] = new String("durga"); → C.E:- Incompatable types
                                      found : String
                                      required : Number.
```

② Object[] a = new Object[10];

```
✓ a[0] = new Object();
✓ a[1] = new Integer(10);
✓ a[2] = new Double(10.5);
✓ a[3] = new String("durga");
```



Case(3):-

→ In the Case of abstract class type arrays as array elements we Can provide its child ~~Cas~~ Class Objects.

Eg1. ① Number[] n = new Number[10];

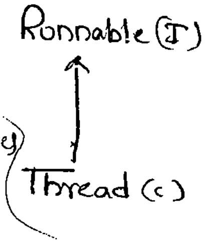
```
✓ n[0] = new Integer(10);
✗ n[1] = new Number();
```

Case 4!-

→ In The Case of Interface type array, as array element we Can provide it's implementation class Objects

Eg!- Runnable[] r = new Runnable[10];

```
  r[0] = new Thread();
X r[1] = new String("durga"); C.E!- Incomptatabletype
                              found: String
                              Required: Runnable
```



Note:-

Array-type	Allowed element-type
1. primitive-type arrays	Any type which Can be implicitly promoted to declared type.
2. Object type arrays	Either declared type Objects or it's child class Objects
3. abstract class type arrays	It's child class objects are allowed.
4. Interface type arrays	it's implementetion class Objects are allowed

Array Variable Assignment :-

Case(i):

→ Element level promotions are not applicable at array level

Eg:- A char value can be promoted to into type. But Char array (char[]) Can't be Promoted to int[] type.

```
① int[] a = {10,20,30,40};
   char[] ch = {'a','b','c'};
 ✓ int[] b = a;
 ✗ int[] c = ch;   C.E:- Incompatiable type
                         found : char[]
                         required : int[]
```

Q) Which of the following promotions are Valid.

✓ ① char → int

✗ ② char[] → int[]

✓ ③ int → long

✗ ④ int[] → long[]

✗ ⑤ long → int

✗ ⑥ long[] → double[]

✓ ⑦ String (child) → Object (Parent)

✓ ⑧ String[] → Object[]

eg:. Child type array we can assign to the parent type variable

→ child-type array we can assign to the parent type variable.

Egs: String[] s = {"A", "B", "C"};

✓ Object[] a = s;

Case(2):-

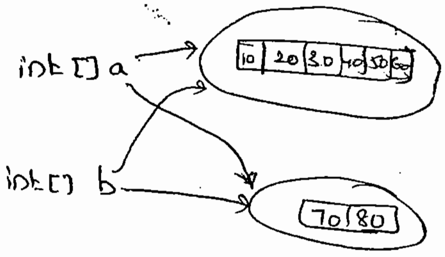
→ when ever we are assigning one array to another array only reference variables will be reassign but not underlying elements. Hence types must be matched but not sized.

Eg:- Case ① int[] a = {10, 20, 30, 40, 50, 60};

int[] b = {70, 80};

✓ ① a = b;

✓ ② b = a;



Eg(2).

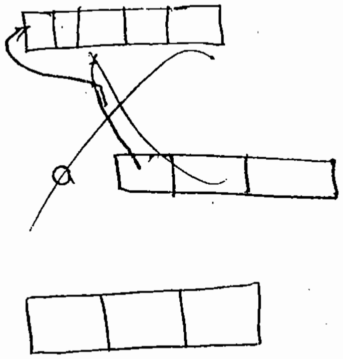
int[][] a = new int[3][2];

a[0] = new int[5];

a[1] = new int[4];

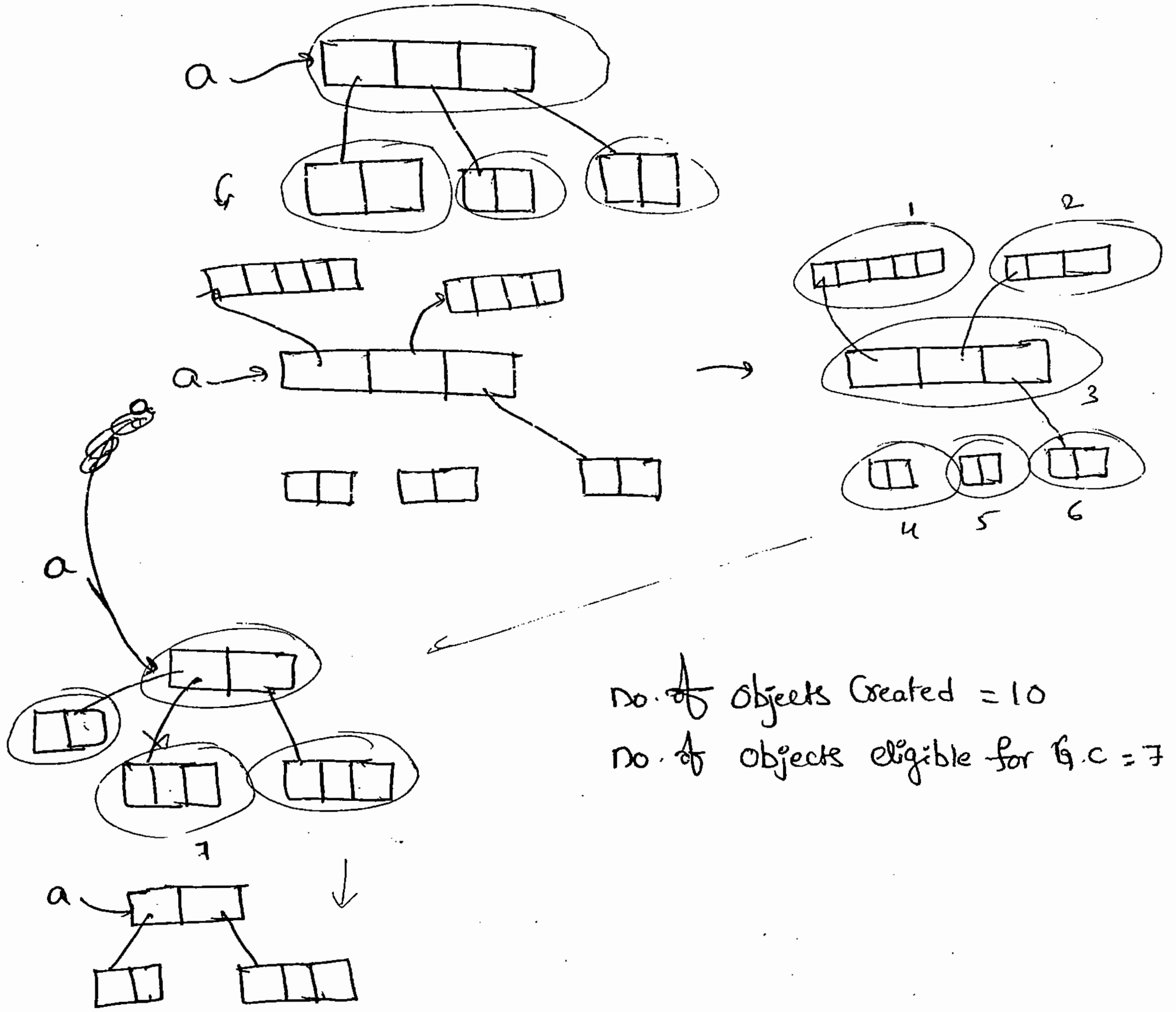
a = new int[2][3];

a[0] = new int[2];



No. of objects Created = 10

No. of objects eligible for G.C = 7.



No. of objects Created = 10

No. of objects eligible for G.C = 7

Case 3:-

→ when ever weare performing array assignments dimenshions must be matched, i.e in The place of Single dimenshional int[] array, ~~Only~~ we should provide only Single dimenshional int[]. by mistake we are providing any other dimenshion we will get Compile time Error

eg:-

```
int[][] a = new int[3][];
    a[0] = new int[3];
    a[0] = new int[3][2]; ⟶ C.E: incompatible types
                                found : int[]()
                                required : int[]
    a[0] = 0;
```

a[0] = 10; C.E:- incompatible types
found : int
required : int[]

Types of Variables

→ Based on the type of value represented by a variable, all variable are divided into 2 types.

(i) primitive variables
(ii) reference variables

(i) Primitive Variables:-

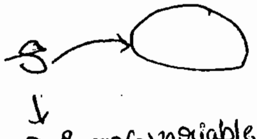
→ Can be used to represent primitive values

ex:- int x = 10;

(ii) reference variables:-

→ Can be used to refer Objects

ex:- Student s = new Student();



s is referencevariable

→ Based on the purpose & position of declaration all variables are divided into 3 types.

(i) instance variables
(ii) static variables
(iii) local variables.

(i) instance variable :-

→ If the value of a variable is varied from Object to object such type of variables are Called instance variable.

→ for every Object a Seperate Copy of instance variable will be Created.

→ the Scope of instance variables is exactly Same as the Scope of the Objects. because Instance variables will be Created at the time of Objects Creation & destroy at the time of Objects destruction.

→ Instance variables will be Stored as the part of Objects.

→ Instance variables should be declare with in the class directly, But outside of any method or Block or Constructor.

→ instance variables Cannot be accessed from ~~insta~~ Static area directly we Can access by using Object reference.

→ But from instance area we Can access instance members directly

Ex!.

```
Class Test
{
 int x=10;
 p.s.v.m (String[] args)
 {
  S.o.pln(x); → C.E:- non-static variable x Cannot
                      be referenced from Static Context"
```

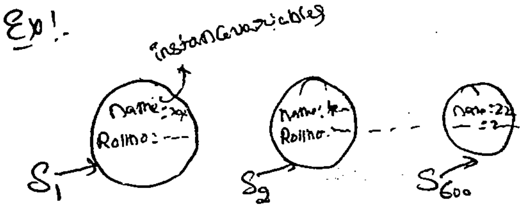
```
    Test t = new Test();
    S.o.pln(t.x); 10 ✓
   }
   public void m1()
   {
     S.o.pln(x); ✓10
   }
  }
```

→ for the instance variables it is not required to perform initialization Explicitly, Jvm will provide default values.

Eg:-

```
  class Test
  {
    String s;
    int x;
    boolean b;
  p.s.v.m(String[] args)
   {
    Test t = new Test();
    S.o.pln(t.s); null
    S.o.pln(t.x); 0
    S.o.pln(t.b); false
   }
  }
```

Ex!-



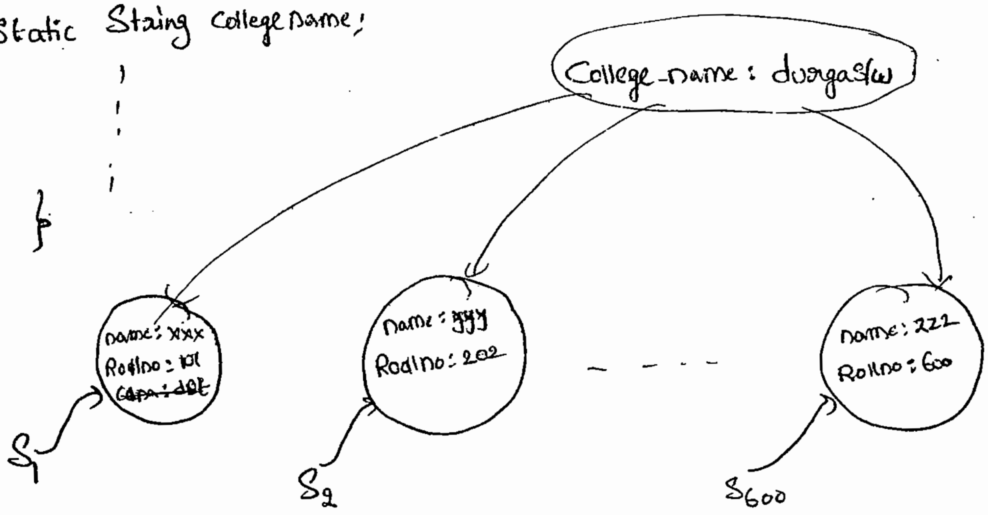
Students objects, In that name, Rollnos are instance variables, Bcz, These values are varied from object to Object.

→ instance variables also known as "Object level variables" or "attributes".

(ii) Static Variables :-

ex:-

```
Class Student
{
    String name;
    int rollno:
 Static String collegename;
        |
        |
        |
}
```



↳ If the value of a variable is not varied from Object to Object then it is never recommended to declare that variable at Object Level we have to declare Such type of Variables at class Level by using Static modifier.

→ In the Case of instance variables for every Object a Seperate Copy will be Created, But in the Case of Static Variable Single Copy will be Created at Class Level & The Copy will be Shared by all Objects of that class.

→ Static variables will be Created at the time of Class Loading & destory at the time of Class unloading. Hence the Scope of the Static variable is

Exactly Same as the Scope of the class.

Note:- Java Test ↲ execution process is

1. Start jvm
2. Create main Thread
3. Locate Test.class
4. Load Test.class → Static variables Creation
5. Execute main() method of Test.class
6. unload Test.class → Static variables destruction
7. Destroy main Thread
8. ShutDown Jvm

→ Static variables should be declare with in the class directly (but Outside of any method or Block or Constructor), with Static-modifier.

→ Static variables Can be accessed either by using class name or by using Object reference, but recommended to use Class name.

→ With in the Same class even it's not required to use Class name. also we Can access directly.

Ex:-

```
class Test
{
    Static int x =10;
    p.s.v.main(String[] args)
    {
        S.o.pln(Test.x); ✓10
        S.o.pln(x); ✓ 10
        ✓Test t = new Test();
        S.o.pln(t.x); ✓ 10
    }
```

→ Static variables are Created at the time of Class loading. i.e, (at the begining of the program). Hence, we Can access from both instance & Static areas directly.

→ Eg:

```
Class Test
{
  Static int x=10;
  p.s.v.m(String[] args)
  {
    S.o.pln(x);
  }
  Public void m1()
  {
    S.o.pln(x);
  }
}
```

→ For the Static variables it is not required to perform initialization Explicitly, Compulsary Jvm will provide default values.

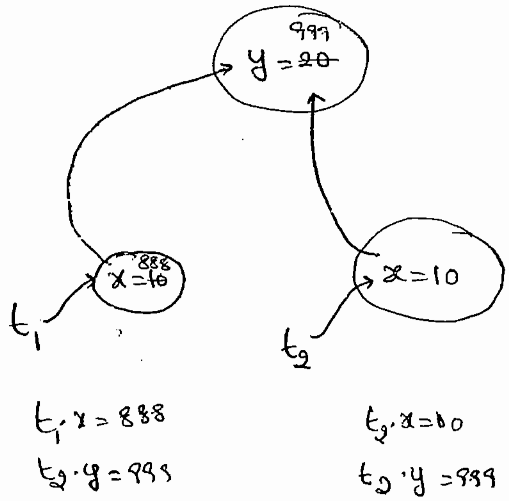
eg:-

```
Class Test
{
  Static int x;
  p.s.v.m(String[] args)
  {
    S.o.pln(x);  0
  }
}
```

→ Static variables will be stored in method-area. Static variables also known as "class-level variables" or "fields".

Ex:-

```
Class Test
{
   int x=10;
  Static int y=20;
  p.s.v.m(String[] args)
  {
   Test t1=new Test();
     t1.x =888;
      t1.y =999;
   Test t2=new Test();
  S.o.pln(t2.x + "----"+t2.y);
  }        10            999
}
```



↳ If we performing any change for instance variables these changes wont be reflected for the remaining objects. because, for every object a seperate copy of instance variables will be their.

→ But, if we are performing any change to the static variable, these changes will be reflected for all objects because we are maintaining a single copy.

(iii) Local Variables:-

→ To meet temparary requirements of the programmer Some-times we have to Create Variables inside method or Block or Constructor. Suchtype of variables are Called Local variables.

→ Local variables also known as Stack variables or Automatic Variables or temparary variables.

→ Local variables will be Stored inside a Stack.

→ The Local variables will be Created while Executing the block in which we declared it & destoryed once the Block Completed. Hence, The Scope of Local Variable is Exectly Same as the Block in which we declared it.

Ex:-

```
Class Test
{
 p.s.v.m(String[] args)
 {
  int i=0;
  for(int j=0; j<3; j++)
  {
   i = i+j;
  }
  S.o.pln(i+"----"+j);
 }
}
```

*C.E:-
Can't find Symbol
Symbol : variable j
Location: Class Test

→ for the Local variables JVM won't provide any default values, Compulsory we should perform initialization Explicitly, before using That Variable.

Eg:- ①

```
Class Test
{
 p.s.v.m(String[] args)
 {
  int x;
✓ S.o.pln("Hello");
 }
}
```

O/P:- Hello

```
class Test
{
 p.s.v.m(String[] args)
 {
   int x;
   S.o.pln(x);
 }
}
```

C.E:-

Variable x might not have been initialized.

eg(2):-

```
Class Test
{
 p.s.v.m(String[] args)
 {
  int x;
  if(args.length >0)
  {
   x = 10;
  }
  S.o.pln(x);
 }
}
```

C.E:- Variable x might not have been initialized

Eg 3:-

```
Class Test
{
 p·s·v·m(String[] args)
 {
  int x;
  if(args.length >0)
  {
   x=10;
  }
  else
  {
   x=20;
  }
  S.o.pln(x);
 }
}
```

O/P:-

```
Java Test ↵
 20
java Test x y ↵
 10
```

→ Note:-

→ It is not recommended to perform initialization of Local variables inside logical blocks because ~~there~~ is no garantee execution of these blocks at runtime.

→ It is highly recommended to perform initialization for the local variables at the time of declaration, at least with default values.

→ The only applicable modifier for the local variables is "final".

If we are using any other modifier we will get Compile-time Error.

Eg:-

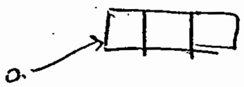
```
Class Test
{
 p.s.v.m(String[] args)
 {
  ✗ private int x=10;   }
  ✗ public int x=10;    }  C.E:- Illegal Start of Expression.
  ✗ Protected int x=10; }
  ✗ Static int x=10;    }
  ✓ final int x=10;
 }
}
```

Un Initialized Arrays:-

```
Class Test
{
int[] a;
 p.s.v.m(String[] args)
 {
  Test t1 = new Test();
   S.o.pln(t1.a);   null
   S.o.pln(t1.a[0]);  NullpointerException
 }
}
```

instanc level :-

```
int [] a ;               S.o.p(obj.a)       null
 i.e   a=null            S.o.p(obj.a[0])    NullpointerException

int[] a = new int[3];    S.o.p(obj.a)       [I@1a2b3
                         S.o.p (obj.a[0])   0
```



Static level :-

```
Static int[] a;          S.o.p(a);   null
                         S.o.p(a[0]); NPE

Static int[] a = new int[3];   S.o.p(a); [I@1234
                               S.o.p(a[0]); 0
```

Explanation :-

int[] a; → here the array (i.e object) reference is Create but its not initialized (i.e object is not) Created. So JVM provides null value to the Variable a.

int[] a = new int[3]; → here becoz of new operator we are Creating an Object and jvm by default provides '0' value in array

Local Level :-

```
int[]a ;                 S.o.p (a)     }  C.E :-  variable a might not have
                         S.o.p(a[0])   }          been initialized
int[] a = new int[3];    S.o.p(a)       [I@1234
                         S.o.p(a[0])    0
```

Note :- Once an Array is Created all its elements are always initialized with default values irrespective weather it is Static or instance or Local array.

8/03/10

Var-arg methods (1.5 version)

→ untill 1.4 version we can't declare a method with variable no. of arguements, if there is any change in no. of arguements compulsary we should declare a new method. This approach increases length of the code & reduces readability.

→ To resolve these problem sun people introduced var-arg methods in 1.5 version. Hence from 1.5 version onwards we can declare a method with variable no. of arguements such type of methods are called var-arg methods.

→ We can declare var-arg method as follows.

$\boxed{\text{m1(int... x)}}$

→ We can invoke this method by passing any no. of int values including zero no. also.

Ex:-
```
m1();  ✓
m1(10, 20);  ✓
m1(10);  ✓
m1(10, 20, 30, 40);  ✓
```

Ex(1):-
```
Class Test
{
 p.s.void m1(int... i)
 {
  S.o.pln("var-arg method");
 }
 p.s.v.m(String[] args)
 {
  m1();
  m2(10);
  m3(10, 20);
  m4(10, 20, 30, 40);
 }
}
```

O/p :-
```
var-arg method
var-arg method
 "     "
 "     "
```

→ Internally Var-arg method is implemented by using single dimenshional arrays concept. Hence with in the Var-arg method we can differenciate arguements by using index.

Ex:-

```
Class Test
{
 public static void sum(int... x)
 {
  int total = 0;
  for(int y: x)
  {
   total = total+y;
  }
  S.o.pln(" The sum: "+ total);
 }
 p.s.v.m(String[] args)
 {
  sum();              0
  sum(10,20);          30
  sum(10,20,30)          60
  sum(10,20,30,40);        100
  }
 }
```

O/P:

```
The sum : 0
The sum : 30
The sum : 60
The sum : 100
```

Case(1):-

Q) which of the following var-arg method declarations are valid.

```
m1(int... x) ✓
m1(int x...) X
m1(int ...x) ✓
m1(int. ..x) X
m1(int .x..) X
```

Case 2:-

→ we can mix var-arg parameter with normal parameter also.

ex:- m1(int x, String... y) ✓

Case 3:-

→ If we are mixing var-arg parameter with general parameter then var-arg parameter should be last parameter.

ex:- m1(~~int~~... x, String y) X

Case 4:-

→ In any var-arg method we can take only one var-arg parameter.

ex:- m1(int... x, String... y) X

Case 5:-

```
class Test
{
 p.s.v.m1(int i)
 {
  S.o.pln("General method");
 }
 p.s.v.m1(int... i)
 { S.o.pln("var-arg");
 }
```

```
p.s.v.m(String [] args)
{
  m1(); var-arg
  m1(10); General (only)
  m1(10, 20); var-arg
} }
```

→ In General var-arg method will get Least Priority i.e if no other method matched. Then only var-arg method will get chance. This is Similar to default case inside Switch.

Case 6 :-

Ex :-
```
Class Test
{
 p-s-v-m1(int[] x)
  {
   s-o-pln("int[]");
  }
 p-s-v-m1(int... x)
  {
   s-o-pln("int...");
  }
}
```

C.E :- Cannot declare Both m1(int[]) and m1(int...) in Test.

Var-arg Vs Single dimenshional arrays :-

Case(1) :-

→ Whereever Single dimenshional array present we Can replace with var-arg Parameter.

Ex :- m1(int[] x) ⟹ m1(int... x) ✓

main(String[] args) ⟹ main(String... x) ✓

Case2 :-

→ where ever var-arg parameter present we Can't replace with Single dimenshional array.

✗ m1(int... x) ⟹ m1(int[] x)

09/03/11

main()

main() :-

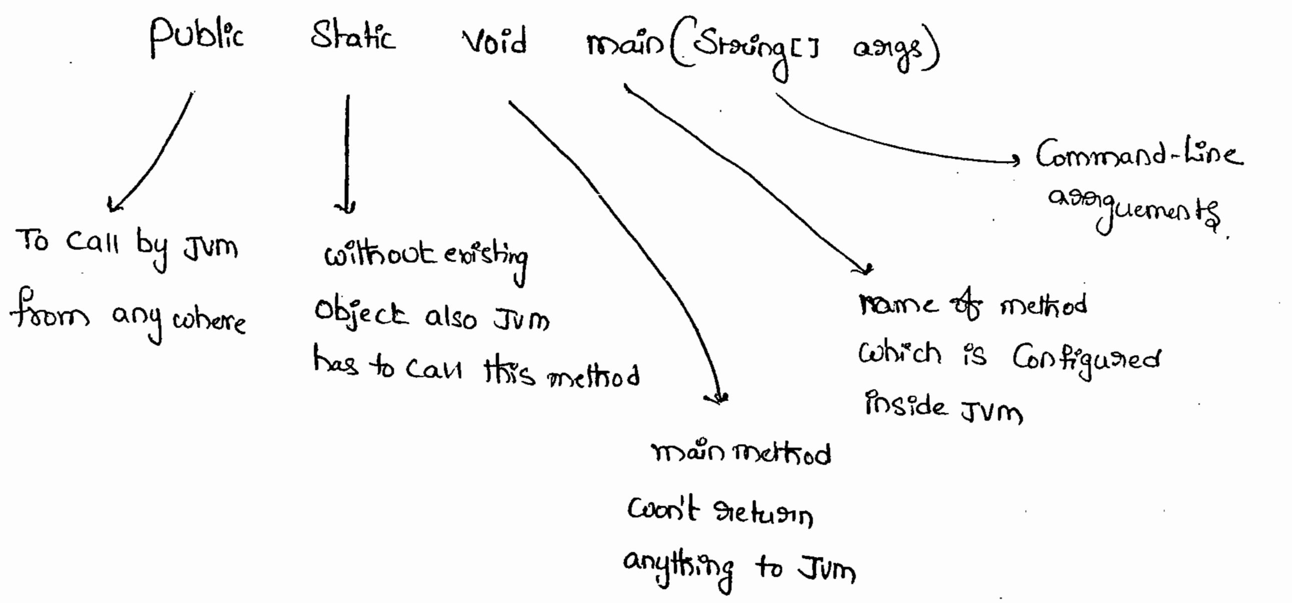
→ wheather the class contains main() or not & wheather the main() is properly declared or not, these checkings are not responsibilities of compiler. At runtime, JVM is responsible for these checking.

→ If the JVM unable to find required main() then we will get runtime Exception saying NoSuchMethodError: main.

Ex:-

```
        class Test
        {
        }
compile Javac Test.java ✓
run ✗  Java Test → R.E:- NoSuchMethodError: main
```

→ JVM always searches for the main() with the following signature.



public static void main(String[] args)

To call by JVM from any where
without existing object also JVM has to call this method
main method can't return anything to JVM
name of method which is configured inside JVM
Command-Line arguements

→ If we are performining any change to the above Signature we will get Runtime Exception Saying " NoSuchMethodError: main ".

→ Any where The following changes are acceptable.

(1) we Can change The Order of modifiers. i.e instead of public static weCan take static public.

(2) We Can declare String[] in any valid form

String[] args ✓
String []args ✓
String args[] ✓

(3) Instead of args we Can take any valid Java identifier.

(4) Instead of String[] we Can take Var-arg String parameter. is String...

main(String[] args) ⟹ main(String... args) ✓

(5) main() Can be declared with the following modifiers also

(i) final
(ii) Synchronized
(iii) Strictfp.

Ex:

```
class Test
{
 final static strictfp synchronized public void main(String... A)
 {
  S.o.pln(" Hai durga");
 }
}
```

Q) which of the following main() declarations are valid?

Ans:- (i) public static int main(String[] args) ✗

(ii) static public void Main(String[] args) ✗

(iii) public synchronized strictfp final void main(String[] args) ✗

(iv) public final static void main(String... args) ✗

✓ (v) public strictfp synchronized static void main(String[] args)

Q) In which of the above cases we will get Compiletime Error.

Ans:- No where, All cases will Compile.

→ Inheritance Concept is applicable for static methods including main() also. Hence if the child class doesn't contain main() then Parent class main() will be executed while executing child class.

Ex:-

```
class P
{
 public static void main(String[] args)
 {
   S.opln("ILU durga s/w");
  }
}
class C extends P
{

}
javac P.java ✓
java P
O/P:- ILU durga s/w
java C
O/P:  ILU durga s/w
```

ex 2:-

```
Class P
{
 p.s.v.m(String[] args)
 {
  S.o.pln(" I Love");
  }
 }
 class  C extends  P
 {
  p.s.v.m(String[] args)
  {
   S.o.pln(" durgas/w");
  }
 }
```

```
Javac  P.java
java  P
o/p!  ILove
java  C
o/p!- durga s/w.
```

→ It Seems to be overriding Concept is applicable for Static methods, but it's not overriding but it is Methodhidding.

→ Overloading concept is applicable for main() but JVM always Calls String[] ~~array~~ arguement method only. The other method we have to Call Explicitly.

ex!-

```
class Test
{
 p.s.v.m(String[] args)
 {
 S.o.pln(" durgas/w");
 }
 p.s.v.m(int[] args)
 {
 S.o.pln(" is good");
}}
```

o/p :- durgas/w.

Q) Instead of main is it possible to Configure any other method as main method?

A) Yes, But inside Jvm we have to Configure Some changes then it is possible.

Q) Explain about S.o.pln?

A)

```
Class Test
{
 Static String name = "durga";
}
```

Test.Name.length()

It is a Class-name

Static variable of type String present in Test class

it is a method present in String class

```
Class System
{
  Static PrintStream out;
}
```

System.out.println()

It is a Class Name present in java.lang

Static variable of type printStream present in System Class

It is a method present in PrintStream Class

10/03/11

Commandline Arrguements

Commandline arrguements :-

→ The arrguements which are passing from Commandprompt are Called CommandLine arrguements.

→ The main objective of Commandline arrguements are we can Customize the behaviour of the main().

Ex:- Java Test X Y Z

```
Java Test X Y Z

args[0] ← X
 args[1] ← Y
 args[2] ← Z

args.length ⇒ 3.
```

Ex(1) :-

```
class Test
{
 p.s.v.m(String[] args)
 {
  for(int i=0 ; i<=args.length ; i++)
  {
   S.o.pln (args[i]);
  }
 }
}
```

~~fort~~ o/p !-

```
Java Test ↵
R.E !- AIOBE

Java test X Y ↵
X
Y
R.E !- AIOBE
```

Ex(2):-

→ with in The main(), Commandline argguements are available in String form.

Ex:-

```
Class Test
{
 p.s.v.m(String[] args)
 {
  S.o.pln(args[0] + args[1]);
 }
}

Java Test 10 20
```

O/P:- 1020

→ Space is the Seperater B/w CommandLine argguments, if the Command-Line argguments itself Contain Space then we should enclose with in doubleCodes (")

ex:-

```
Class Test
{
 p.s.v.m(String[] args)
 {
  S.o.pln(args[0]); Note Book
 }
}
       Java Test "Note Book"
```

Ex(3):

```
class Test
{
 p.s.v.m(String[] args)
 {
  String[] argh = {"A", "B"};
  args = argh;
  for(String S1 : args)
  {
```

```
   S.o.pln(S1);
  }
 }
}
```

```
Java Test x y ↵
o/p A
    B

Java Test x y z ↵
o/p A
    B

Java Test ↵
o/p A
    B
```

Note: The maximum allowed no. of Commandline arreguements is 2147483647, min. is '0'

Java Coding Standards

→ whenever we are writing the Code it is highly recommended to fallow Coding Conversions the name of the method or Class should reflect the purpose of functionality of that Component.

```
Class A
{
public int m1(int x, int y)
{
  return x+y;
}
}
```

-Ameerpet Standard

```
package Com.durgasoft.demo;
Public Class Caluclator
{
  Public Static int Sum(int number1,
                        int number2)
  {
    return number1+number2;
  }
}
```

-Hitech-City

Coding standards for classes:-

→ Usually Classnames are Nouns, should starts with UpperCaseLetter & if it Contains multiple words every inner word should starts with UpperCase Letter

Ex:- Student
Customer
String
StringBuffer, } → Nouns

2) Coding Standards for Interfaces :-

→ Usually interface names are Adjectives should starts with UpperCase Letter & if it Contains multiple words every inner word Should starts with UpperCase Letter.

Ex:- Runnable, Serializable, Cloneable, Movable. } Adjectives

Note :-

Throwable is a Class but not interface. It acts as a root Class for all Java Exceptions & Errors.

3) Coding Standards for Methods :-

→ Usually method names are Either Verbs or Verb noun Combination Should starts with Lower Case Letter & if it Contains multiple words "Every inner words should starts with Upper Case Letter". (camelCase).

Ex:
run()
sleep()
eat()
init()
wait()
join()
⋮ } → Verbs

getName()
setSalary() } Verb + noun
⋮

(4) Coding Standards for Variables :-

→ Usually the variable names are nouns Should Starts with LowerCase character & if it Contains multiple words, Every innerword Should starts with uppercase character (Camel Case).

Ex! Name
roll no
mobile Number
⋮
} → nouns

⑤ Coding Standards for Constants:-

→ Usually The Constants are Nouns. Should Contain only UpperCase characters, If it Contains multiple words, These words are Seperated with "–" symbol.

→ We Can declare Constants by using Static & final modifiers.

Ex:-
MAX–VALUE
MIN–VALUE
MAX–PRIORITY
MIN–PRIORITY

⑥ Java bean Coding Standards

→ A Java bean is a Simple java class with private properties & public getter & setter methods.

Ex:-

```
public class StudentBean
{
private String name;
 Public void setName(String Name)
 {
   this.name = name;
 }
 public String getName()
 {
   return name;
  }
}
```

→ Ends with Bean is not affical Convention from SUN.

Syntax for Setter method :-

→ The method name should be prefix with "Set". Compolsary the method should take some arrguement. return type should be void.

Syntax for getter method :-

→ the method name should be prefixed with "get".

→ It should be no arrguement method.

→ returntype should not be void.

** Note :-

→ for the boolean property the getter method can be prefixed with either get or is. recommended to use "is"

Ex:-

```
private boolean empty;

✓  public boolean getEmpty()
   {
     return empty;
   }

✓  public boolean isEmpty()
   {
     return Empty;
   }
```

⑦ Coding Standards for Listeners :-

* To register a Listener :-

→ method name should be prefix with add,

→ after add what ever we are taking the arrguement should be same.

eg:- ✓ ① public void addMyActionListener(MyActionListener l)

✗ ② public void registerMyActionListener(MyActionListener l)

✗ ③ public void add MyActionListener(Listener l)

To unregister a Listener:-

→ The rule is Same as above, Except method name should be Prefix with remove.

eg:- ✓ ① public void removeMyActionListener(MyActionListener l)

✗ ② public void unregisterMyActionListener(MyActionListener l)

✗ ③ public void delete MyActionListener(MyActionListener l)

✗ ④ public void removeMyActionListener(ActionListener l)

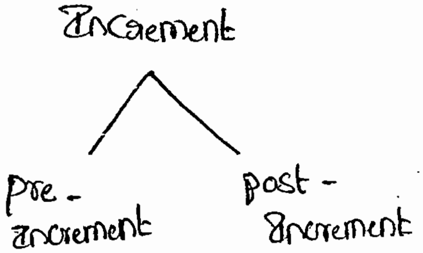
Note:-

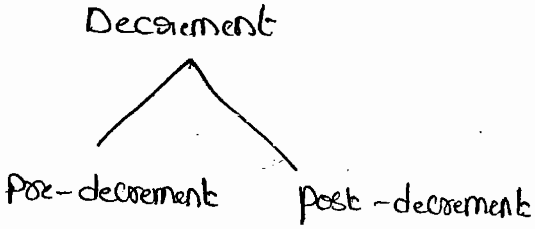
In Javabean Coding Standards & Listener Concept 1 compulsory.

Operators & Assignments

Kathy Sierra 1.6
book for SCJP.

Increment & Decrement Operators:





```
int x = ++y;     int x = y++;          int x = --y;     int x = y--;
```

Expression	Initial value of x	final value of x	final value of y
y = ++x;	4	5	5
y = x++;	4	5	4
y = --x;	4	3	3
y = x--;	4	3	4

i) We can apply increment and decrement only for variables but not for constant values.

```
  int x= 4;
X int y= ++4;          C.E: unExpected type
  S.o.pln(y);
                            found : Value ②
                            required : Variable ①
```

ii) Nesting of increment & decrement operators is not allowed otherwise we will get Compile time Error.

```
  int x = 4;                          C.E:  UnExpected type
X int y = ++(++x);                          ② found : value
              after incre- it is
          -Constant
           Then
  S.op(y);                                  ① Required : Variable
```

iii). We Can't apply increment & decrement Operators for the final variables.

```
Ex(1):- final int x = 4; X        Ex(2):- final int x = 4; X
         x++;                                  x = 5

          C.E:- Can't assign a Value to final Variable x.
```

iv). we Can apply increment and Decrement operators for "Every primitive data type Except Boolean".

```
① ✓ double d = 10.5;              ② ✓ char ch = 'a';
        d++;                              ch++;
     S.op(d); 11.5                     S.op(ch); // b


     ③ boolean b = true;
   X       ++b;                       C.E:-
        S.o.pln(b);                   operator ++ Can't applied to
                                                          boolean.

✓④ int x = 10;
    x++;
    S.o.pln(x); 11
```

Difference b/w b++ & b=b+1 :-

① ✓

```
byte b=10;
b++;
S.o.pln(b); 11
```

② ✗

```
byte b=10
b=b+1;
S.o.p(b);
```

C.E: possible loss of precission
found : int
Required : byte

Exp:- max(int, typeof a, typeof b)
max(int, byte, int)
Res:- int

③ ✓

```
byte b=10
b = (byte) (b+1)
S.o.pln(b);// 11
```

④

```
byte a=10;
byte b=20;
byte c=a+b;
S.o.pln(c);
```

C.E: PLP
f = int
R = byte

Explanation:-

Max(int, typeof a, typeof b)
Max(int, byte, byte)
result is of type : int
∴ found is int but
required is byte

(+, -, *, %, /)

→ whenever we are performing any arithmetic operation between two variables a & b the result type is always,

$$\boxed{\text{Max (int , typeof a, typeof b)}}$$

```
byte b=10;
b =(byte) (b+1);
S.o.p(b); // 11
```

→ In the Case of Increment & decrement operators the required type casting (internal type casting) automatically performed by the Compiler.

byte b++ ; ⟹ b = (byte) (b+1);

b++ ; ⟹ b = (type of b) (b+1);

Arithematic operators:-

→ The Arithmetic operations are (+, -, *, /, %)

→ If we are applying any Arithematic operator b/w two variables a and b the result type is always.

Max (int, type of a, type of b)

byte + byte = int

byte + short = int

int + long = long

long + float = float

double + char = double

char + char = int

S.o.pln (10 + 0.0); // 10.0

S.o.pln ('a' + 'b') ; 195

S.o.pln (100 + 'a') ; 197.

Infinity:-

→ In the Case of integral arithematic (int, short, long, byte), there is no way to represent infinity. Hence, if the infinity is the result we will always get ArithematicException. (AE : / by zero)

eg:-

S.o.pln (10/0) ; R.E : AE: / by zero.

→ But in Case of floating point arithematic (float & double), there is always a way to represent infinity. for this float & Double Classes Contains the following two Constants.

Positive_Infinity = Infinity
Negative_Infinity = - Infinity

+ve-∞	= ∞
-ve-∞	= -∞

→ Hence, in the Case of ~~float~~ floating point Arithematic we won't get any Arithematic Exception.

Eg:- ①. S.o.pln (10/0.0) ; Infinity

②. S.o.pln (-10/0.0) ; -Infinity.

* NaN :- (Not a Number)

→ In integral arithematic there is no way to represent undefined results. Hence, if the result is undefined we will get A.E in Case of integral Arithematic.

Eg:- S.o.p(0/0) ; R.E: A.E: / by zero

→ But in Case of floating point Arithematic, there is a way to represent undefined results for this float & Double classes Contains NaN Constance.

→ Hence, Eventhough the result is undefined we won't get any Runtime Exception in floating point Arithematic.

Eg:- S.o.pln(0/0.0) ; NaN.

• S.o.p(0.0/0) ; NaN

• S.o.p (-0/0.0) ; NaN

Ex: * public Static ~~v~~double Sqrt (double d);

S.o.pln (math. Sqrt (4)) ;/2.0

S.o.pln (math. Sqrt (-4)) ; NaN.

→ for any x value including NaN the below Expressions always returns false, Except the (!=) Expression returns true.

$$\boxed{X \mathrel{!}= NaN \Rightarrow True}$$

$$\left.\begin{array}{l} x > NaN \\ x >= NaN \\ x < NaN \\ x <= NaN \\ x == NaN \end{array}\right\} false$$

$\boxed{\text{at } x=10}$

S.o.p (10 > float.NaN); false

S.o.p (10 < float.NaN) ; false

S.o.p (10 == float.NaN); flase

~~floa~~ S.o.p (10 != Float.NaN); true.

S.o.p (float.NaN == float.NaN); false

S.o.p(Float.NaN != float.NaN); True.

Conclusion about A.E (AirthmeticException) :-

→ It is RuntimeException but not Compiletime Error.

→ possible only in Integral Arithematic but not floating point Arithmatic

(int, byte, short, char) (float, double)

→ The only operators which cause A·E are / and %.

3. String Concatination Operator (+)

→ the only overloaded operator in Java is '+' operator.

→ Some times it acts as arithematic addition operator & Some time acts as String arithematic Operator (or) String Concatination operator.

Eg:- int a =10, b=20, c=30;

String d = "shanth";

S.o.p(a+b+c+d); 60shanth

S.o.p(a+b+d+c); 30shanth30

S.o.p(d+a+b+c); shanth102030

S.o.p(a+d+b+c); 10shanth2030.

d+a+b+c
shanth10+b+c
shanth1020+c
shanth102030

→ If at least one operand is String type then '+' operator acts as Concatination, otherwise, (if both are number type) '+' acts as arithematic operator.

Here S.o.p() is avaluated from Left to Right.

Eg:- int a=10, b=20;

String c = "shanth";

× $a = \underbrace{b+c}_{\text{total String}}$ C.E:- Incompatible type : found : String
Required : int

✓ $\underset{\text{String}}{c} = \underbrace{a+c}_{\text{total String}}$;

✓ $\underset{\text{int}}{b} = \underbrace{a+b}_{\text{int}}$;

× c = a+b; C.E:- InCompatible type:
found : int
Required : String.

A=65, a=97
5

Relational Operators

These are >, <, >= , <=

1) * we Can apply Relational operators for Every primitive datatype Except boolean.

Eg:-
1) 10 > 20 false ✓
2) 'a' < 'b' True ✓
3) 10 >=10.0 True ✓
4) 'a' < 125 True ✓
5) true <= true
6) true < false

CE:- Operator <= Can't be applied to boolean, boolean

2) * We Can't apply relational operators for the object types.

Eg:- 1) "shanth" < "shanth" ✗ 2) "durga" < "durga123" ✗

CE: operator < can't be applied to String, String.

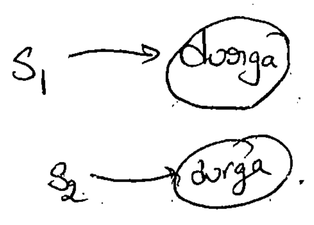
3) * Nesting of Relational operators we are not allowed to apply.

Eg:- ✓ S.o.p (10 < 20);

✗ $S.o.p\,(\underbrace{10<20}_{\text{boolean}}<30)$

CE!- operator < Can't be applied to boolean.

Eg:-
String s1 = new String("durga");
String s2 = new String("durga");



S.o.pln(S1==S2); false (reference)
S.o.pln(S1.equals(S2)); true (Content)

Equality Operators (==, !=)

→ These are ==, !=

** We Can apply Equality operators for Every primitive type including boolean types.

**Eg:-

		o/p	
✓ 1)	10 = = 10.0	T	✓
✓ 2)	'a' = = 97	T	✓
✓ 3)	true = = false	F	✓
✓ 4)	10.5 = = 12.3	F	✓

↔ We Can apply Equality operators even for object reference also.

→ for the two object references r_1 and r_2 {$r_1 == r_2$ returns True iff both r_1 & r_2 are pointing to the same object.

i.e, Equality operator (==) is always ment for reference/address Comparison

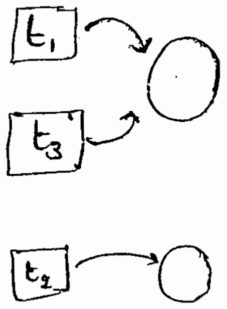
Ex(1): Thread t_1 = new Thread();

Thread t_2 = new Thread();

Thread t_3 = t_1;

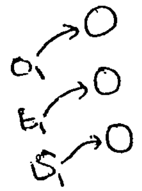
✗ S.o.p (t_1 = = t_2) ; False

✓ S.o.p (t_1 = = t_3) ; True



*→ To apply Equality Operators b/w the object references Compulsory there should be Some relationship b/w arguement types. [either parent to child (or) child to parent (or) Same type] otherwise we will get CE: InComparable type].

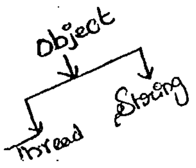
Eg:-(2):- object o1 = new Object(); because object is Super class



Thread t1 = new Thread();

String s1 = new String("shanth");

S.o.p(t1 = = s1); CE:- Incomparable types java.lang Thread & java.lang. String



S.o.p(t1 = = o1); F

S.o.p(s1 = = o1); F

→ for any object reference ~~r~~ x, if r is pointing to any object $\boxed{r == \text{null is always, false}}$, otherwise r contains null value

→ So, $\boxed{\text{null} == \text{null is always True.}}$

*Note:-

* In General, == operator ment for reference Comparision where as .equals() method ment for Content Comparision.

Instanceof operator (instanceof) ✓

↳ By using this operator we can check, whether the given object is ~~of~~ a particular type or not.

Syn:- $\boxed{r \text{ instanceof } X}$

any reference type

class/ interface.

instanceof
Hashtree
Strictfp

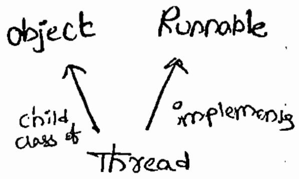
Ex:- short s=15;
Boolean b;
b = (s instanceof Short)
b = (s instanceof Number)

Eg:- ** 1) Thread t = new thread()

✓ S.o.p(t instanceOf Thread) ; True

✓ S.o.p(t instanceOf Object) ; True

✓ S.o.p(t instanceOf Runnable) ; True



↳ To use instanceOf operator, Compulsary there should be Some relationship b/w argument type, otherwise we will get Compile-time Error Saying Inconvertable type.

Eg:- 2) Thread t = new thread();

S.o.p(t instanceof String); C.E:-
Inconvertable type
found : Thread
Required : String

↳ whenever we are checking parent object is of child type Then we will get false as output.

Object o = new ~~Object();~~ Integer(10);

✓ S.o.p(o instanceOf String) ; false

↳ for any class ~~or~~ interface ~~of~~ X, null instanceOf X always returns "false".

✓ S.o.p(null instanceOf String) ; false.

Eg:- Iterator itr = l.iterator();
while (itr.hasnext())
{

Object o = itr.next();
if (o instanceOf Student)
{ Apply Student related function

}
else if (o instanceof Cu)
{
Apply Customer related

Bit-wise Operators :-

(1) & → AND ⇒ if Both operands (or) arguements are True then Result is True

(2) | → OR ⇒ if atleast 1 operand is T " " T

(3) ^ → X-OR ⇒ if Both operands are different " " T

Eg:- S.o.pln(T&T); T

S.o.pln(T|T); T

S.o.pln(T^T); F

Ex(1):- S.o.pln(4&5); 4 →
```
100
101
100 = 4
```

S.o.pln(4|5); 5 →
```
100
101
101 = 5
```

S.o.pln(4^5); 1 →
```
100
101
001 = 1
```

→ We Can apply these Operators Even for integral data-types also. also.

Ex:- (1) S.o.pln(4&5); 4

(2) S.o.pln(4|5); 5

(3) S.o.pln(4^5); 1

Bitwise Complement Operator (~) :- (Filed)

S.o.pln(~T);   CE: operator ~ can't be applied to boolean

(1) We can apply Bitwise Complement Operator only for integral types, but not for boolean type.

Ex:- ✗ 1) S.o.pln(~True);

C.E: operator ~ Can't be applied to boolean.

✓ 2) S.o.pln(~4); -5

```
 4  ≡ 0000 0000 ---- 0100                    0 → +ve
~4  = [1]111 1111 ---- 1011                  1 → -ve
       ↓            ↳ 2's Complement
      -ve

        One's Comp
             000 0000 ---- 0100
                                      add '1' to 1's Comp
  2's Comp              0        1
             ______________________      is 2's Comp
             000 -------- 0101
             ______________________

      -ve 5
```

$\boxed{\therefore -5}$

Note:

→ The most Significant bit represents Sign bit. 0 means +ve no, 1 means -ve no.

→ +ve no. will be represented directly in the memory. where as -ve no's will be represented in 2's Complement form.

Boolean Complement Operator (!) :-

→ we Can apply these operator only for Boolean type but not for integral types.

Ex!- (1) S.o.p(!4);

C.E!- operator ! Can't be applied to int.

(2) S.o.p(! False); True

(3) S.o.p(! True); False

Summary:-

&
|
^ ⇒ we Can apply for both integral & boolean types.

~ ⇒ we Can apply only for integral types but not for boolean types.

! ⇒ we Can apply only for boolean types but not for integral types.

Short-Circuit Operators (&&, ||)

→ double AND
→ double OR

1) We Can use these operators Just to improve performance of the System.

2) these are Exactly Same as normal bitwise operators &, | Except the following difference.

&, |	&&, ||
1. Both operands should be Evaluated always.	1. 2nd operand Evaluation is optional.
2. Relatively Low-performance	2. Relatively High-performance.
3. Applicable for Both Boolean & integral types	3. Applicable only for Boolean types.

Ex:-

```
if( ~~~~~~~   &   ~~~~~ )
      e1      ↓       e2
    10min    1min    10min
            21 min
{
   =
}
else
{
   =
}
```

1) x && y ⇒ y will be Evaluated iff x is True.

2) x || y ⇒ y will be Evaluated iff x is false.

Ex:-

```
int x=10;
int y=15;
if (++x >10 & ++y<15)
{
   ++x;
}
else
{
  ++y;
}
S.o.pln(x+ "------"+y);
```

O/p:-

	x	y
&	11	17
|	12	16
||	12	15
&&	11	17

⑨

```
int x =10;
if ((++x <10) && (x/0 >10))
{
S.o.pln("Hello");
}
else
{
 S.o.pln("Hi");
}
```

Ans:

a) C.E

b) R.E : Arithematic Exception : 1 by Zero.

c) Hello

d) Hi

Note:

if we Replace && with &

then Result is ⓑ, that is R.E.

a=97
A=65

TypeCast Operators:-

→ There are 2 types of primitive type Castings.

1. Implicit type Casting
2. Explicit type Casting.

Implicit TypeCasting:-

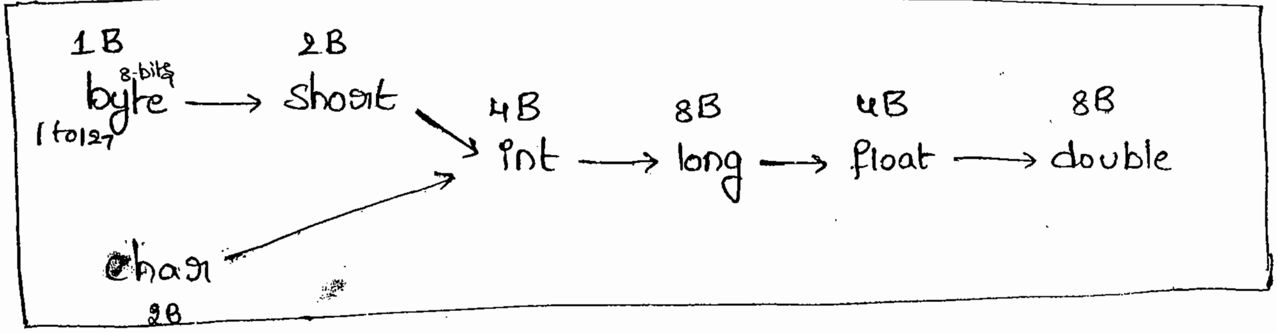
1) Compiler is the responsible to perform this typeCasting

2) This TypeCasting is required when ever we are assigning Smaller data type value to the bigger data type variable.

3) It is also known as " Widening (or) UpCasting".

4) No loss of information in this type Casting.

→ The following are various possible implicit typeCasting



Ex(1)!.

① double d = 10;   [Complier Converts ~~int~~ into double automatically]

/ S.o.pln(d); 10.0

② int x ='a';   [Compiler Converts char to int automatically]

/ S.o.pln(x); 97

a = 97, b = 98 - - -
A = 65, B = 66, C = 67,

2) Explicit type Casting :-

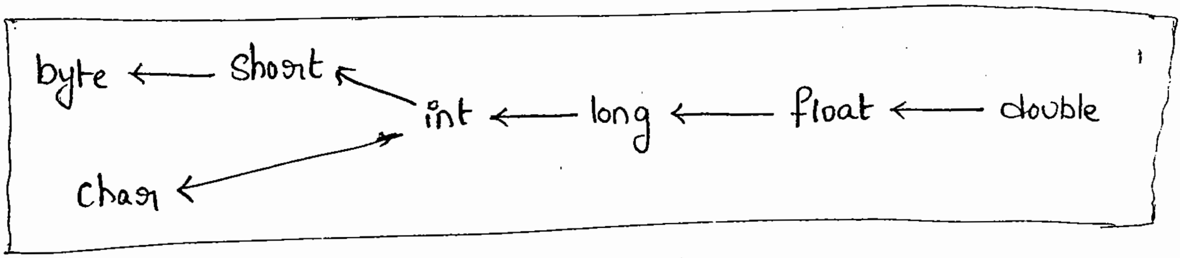
1) programmer is responsible to perform this TypeCasting

2) It is required when ever we are assigning bigger datatype value to the Smaller datatype variable.

3) It is also known as " Narrowing or down Casting".

4) There may be a chance of loss of information in this Type-Casting.

→ The following are various possible Conversions where Explicit typeCasting is required.



Ex !.

1) X | byte b = 130

C.E : possible loss of precission

found : int

Required : byte

2)
```
byte b = (byte) 130;
S.o.p(b);   -126
```

→ when ever we are assigning Bigger datatype value to the Smaller data-type variable then the most Significant bit will be lossed.

① ✗ byte b = 130 ;

✓ byte b = (byte) 130 ;

2|130
2|65 - 0
2|32 - 1
2|16 - 0
2|8 - 0
2|4 - 0
2|2 - 0
2|1 - 0

130 ≡ 0000 - - - - - - - - - $\underline{10000010}$ (32-bits)

byte b ≡ 1 $\underline{0000010}$ (8 bit)

-ve

2's Complement

1111101

$\quad\quad 1$

1111110

0000010

1111110

$= 1\times2^6 * 1\times2^5 + 1\times2^4 + 1\times2^3 + 1\times2^2 * 1\times2^1 + 0\times2^0$

$= 64 + 32 + 16 + 8 + 4 + 2 + 0$

~~+ve~~ 126

$\boxed{\therefore -126}$

② int i = 150 ;

short s = (short) i ;

S.o.pln (s) ~~=~~ 150

150 ≡ 0000 - - - - - 0 1001 0110 (32 bits)

short s ≡ 0000 - - - $\underline{0001 0110}$ ⟶ 2 Bytes = short = 16-bits

+ve

~~d~~don't apply 2's Comp.

∴ S = 150

③ int x = 150 ;

byte b = (byte) x ;

short s = (short) x ;

S.o.pln (b) ; -106

S.o.pln (x) ; 150

150 ≡ 0000 - - - 0 10010110

byte ~~b~~ = 1 $\underline{0010110}$

-ve

2's Com

1101001

$\quad\quad 1$

1101010

1101010

$\boxed{\therefore -106}$ = 2 + 8 + 32 + 64 = 106

10|2|11

→ when ever we are assigning floating point datatype values to the integral data types by Explicit type Casting the digits after the decimal Point will be lossed.

Ex!-

```
double d = 130.456;
int a = (int) d;
byte b = (byte) d;
S.o.pln (a); 130
S.o.pln (b); -126
```

Assignment Operators :-

→ There are 3 types of assignment operators

1. Simple assignment operators
2. chained assignment operator
3. Compound assignment operator.

1. Simple assignment operator :-

Ex!- `int x = 10;`

2. chained assignment operator :-

Ex!-

```
int a, b, c, d;
a = b = c = d = 20;
```

| ✓

→ We Can't perform chained assignment at the time of declaration

Ex:- int a = b = c = d = 20 ; } X C.E

```
C.E: Can't find Symbol
     Symbol : variable b
     location : class Test
                 int a = b = c = d = 20 ;
                         ^
                                      (Same as c, & d)
```

Ex:- ② int b, c, d;

a = b = c = d = 20 } ✓

3. Compound assignment operator :-

→ Some times we can mix assignment operator with some other operator to form Compound assignment operator.

```
Ex:-  int a = 10 ;                a += 30
        a += 30 ;                 a = a + 30
      S.o.pln (a); 40             a = 10 + 30
                                    a = 40
```

→ The following are various possible Compound assignment operators in Java.

```
  +=   |  &=   |  >>=
  -=   |  |=   |  >>>=      (11)
  %=   |  ^=   |  <<=
  *=   |       |
  /=   |       |
```

✱ In Compound assignment operators the required typecasting will be performed automatically by the Compiler.

Ex ①

✗
```
byte b=10;
b = b+1;
S.o.pln(b);
```

C.E: PLP

found : int

required : byte

$b = b \underset{\wedge}{+} 1;$

✓
```
byte b=10;
b++;
S.o.pln(b); 11
```

```
byte b=10
b+=1 ;
S.o.pln(b) ; 11
```
✓

```
byte b=127;
b += 3;
S.o.pln(b) ; -126
```
✓

Ex ② :-

```
int a, b, c, d;
a=b=c=d=20;
a += b *= c += d /= 2;
S.o.pln(a+"----"+b+"----"+c+"-----"+d);
        620        600       30        10
```

Conditional Operator (?:)

→ The only ternary operator available in Java is a Ternary Operator (or) Conditional Operator.

a+b → binary operator

++a → unary "

(a+b)? a:b; → ternary.

Ex :-

```
int a = 10, b=20;
int x = (a>b) ? 40:50;
S.o.pln(x) ; 50
```

a>b is T then 40

a>b is F then 50

→ nesting of Conditional operator is possible.

Ex:- int a=10, b=20;

```
int x = (a>50) ? 777 : ((b>100) ? 888 : 999));
           F                F
S.o.pln(x) ; 999
```

Ex:- int a=10, b=20;

```
✓ | byte c = (True) ? 40 : 50;
  | byte c = (False) ? 40 : 50;
```

✓ a<12 T

✗ a<b ✗ C.E

don't Compare these variables

```
✗ | byte c = (a<b) ? 40 : 50;
  | byte c = (a>b) ? 40 : 50;
```

C.E:- PLP

found : int

required : byte.

```
final int a=10, b=20;

✓ | byte c = (a<b) ? 40 : 50;
  | byte c = (a>b) ? 40 : 50;
```

New Operator :-

→ We Can use this Operator for Creation of objects.

→ In Java there is no Delete operator. because distraction of useless object is responsibility of Garbage Collector.

[] operator :-

→ we Can use these Operator for declaring & Creating arrays.

Operator precidence :-

1. Unary operators :-

```
[ ] , x++ , x--
++x , --x , ~ , !
new , <type> (used to type cast)
```

2. Arithematic Operators :-

```
* , / , %
+ , -
```

3. shift operator :-

```
>>> , >> , <<
```

4. Comparision operator :-

```
< , <= , > , >= , instanceof
```

5. Equality operator :-

```
== , !=
```

6. Bitwise operators :-

```
&
^
|
```

7. Short - Circuit operators :-

```
&&
||
```

8. Conditional operators :-

```
?:
```

9. Assignment operators :-

```
= , += , -= , .....
```

Evalution Order of operands :-

→ there is no precidence for operands before applying any operator all operands will be evaluated from left to right.

Ex:-

```
class EvaluationOrderDemo
{
 p.s.v.m (String[] args)
 {
  S.o.p (m1(1) + m1(2) * m1(3) + m1(4) * m1(5)/m1(6));
 }
 p.s. int m1(int i)
 {
  S.o.pln(i);
  return i;
 }
}
```

o/p:- 10

$1 + \underline{2 * 3} + 4 * 5/6$

$1 + 6 + \underline{4 * 5}/6$

$1+6+20/6$

$1+6+3$

$7+3$

$=10$

Ex(2) !-

```
class Test
{
 p.s.v.m (String[] args)
 {
  int x =10;
   x = ++x;
  S.o.pln(x); 11
 }
}
```

1st increment

2nd place init into x

```
int x =10 ;
 x = x++ ;
S.o.pln(x) ; 10
```

1st place x =10

∴ x =10++

↳ x =11

but last operation is

x=10

Ex (3) !-

int x =0 ;

$(1+2)^3$

$x = \underbrace{\overset{\downarrow}{++x}}_{1} + \underbrace{\overset{①}{x++}}_{1} + \underbrace{\overset{②}{x++}}_{2} + \underbrace{++x}_{4};$

S.o.p(x) ; 8

$x = \cancel{0}\cancel{1}\cancel{2}\cancel{3}4$

x++ = 1

x++ = 2

3

4

Ex 4 !-

int x=0 ;

x += ++x + x++;

S.o.pln(x) ; 2

x = x+ ++x + x++;

= 0 + 1 + 1

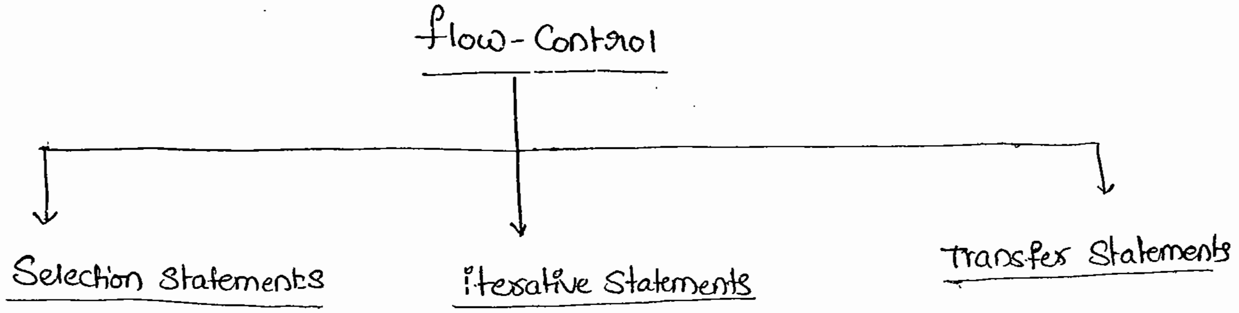
x = 2

Flow Control

16/05/2011

Flow Control :-

→ Flow Control describes the order in which the statements will be executed at runtime.



Selection statements

① if-else
② Switch

Iterative Statements

① while
② do-while
③ for()
④ for-each loop

Transfer Statements

① break
② Continue
③ return
④ try
⑤ catch
⑥ final

(a) Selection Statements:-

(i) if-else :-

Syn :-

```
if(b)
{
  Action if b is true
}
else
{
  Action if b is false
}
```

1→ The arguement to the if Statement should be boolean type. if we are providing any other type we will get Compiletime Error.

Ex:-

①
```
int x=0
if(x)
{
S.o.pln("Hello");
}
else
{
S.o.pln("Hi");
}
}
```

C.E:- incompatable types
found : int
required : boolean

②
```
int x=10
if(x =20)
{
S.o.pln("Hello");
}
else
{
S.o.pln("Hi");
}
}
```

③
```
int x =10;
if(x = =20)
{
S.o.pln("Hello");
}
else
{
S.o.pln("Hi");
}
}
```

✓

o/p !- Hi

④
```
boolean b= false;
if(b =true)
{
S.o.pln("Hello");
}
else
{
S.o.pln("Hi");
}
}
```

✓

o/p!. Hello

⑤
```
boolean b= false;
if (b = = true)
{
S.o.pln(" Hello");
}
else
{
S.o.pln(" Hi");
}
}
```

✓

o/p!- Hi

(2) Curlly brases ({ . }) are optional and without Curllybrases we can take only one Statement & which should not be declarative Statement

Ex:-

```
if(true)
 S.o.pln("Hello");
```
✓

```
if(true)
 int x=10;
```
✗
C.E!

```
if(true)
{
 int x=10;
}
```
✓

```
if(true);
```
✓

Switch Statement :-

→ If Several options are possible then it is never recommended to use if-else, we should go for Switch Statement.

Syn:-

```
Switch(x)
{
 Case1 :
         Action1;
 Case 2:
         Action 2;
   :
 default :
         default Action;
}
```

↦ Curlly brases are mandatory.

→ both Case & default are optional inside a Switch

Ex:-

```
int x=10;
Switch(x)
{
}
```
✓

→ with in the Switch, every Statement should be under some Case or default. Independent Statements are not allowed.

Ex:-

```
int x=10;
Switch (x)
{
  S.o.p("Hello");
}
```

C.E:-
Case, default or '}' expected

→ until 1.4v the allowed datatypes for switch arguement are

byte
Short
int
Char

→ But from 1.5v onwards in addition these the Corresponding wrapper classes (Byte, Short, Character, Integer) & enum types are allowed.

1.4 v	1.5v	1.7v
byte	+ Byte	
Short	Short	+ String
char	Character	
int	Integer	
	+	
	enum	

→ if we are passing any other-type we will get Compiletime Error.

Ex:-

```
byte b=10;
switch (b)
{
}
✓
```

```
Char ch='a';
Switch (ch)
{
}
✓
```

```
long l=10l;
Switch(l)
{
} X
```

C.E:-
possible loss of precision
found: long
required: int

```
boolean b=true;
Switch(b)
{
}
```

C.E:-
Incompatible types
found: boolean
required: int

→ every Case label should be within the range of Switch arguement type otherwise we will get Compiletime Error.

Ex:

```
byte b=10;
Switch(b)
        ↳ byte
{
 Case 10:
       S.o.pln("10");
 Case 100:
       S.o.pln("100");
 Case 1000:                -128 to
       S.o.pln("1000");      127
}
```

C.E:- possible Loss of precision
found: byte int
required: byte.

```
byte b=10;
Switch(b+1) → int type
{
 Case 10:
       S.o.pln("10");
 Case 100:
       S.o.pln("100");
 Case 1000:
       S.o.pln("1000");
}
                    ✓
```

→ every Case label should be a valid Compile-time Constant, if we are taking as Variable as Case label we will get Compiletime Error.

Ex:-

```
int x=10;
int y=20;
Switch(x)
{
 Case 10:
        S.o.pln("10");
 Case y:
        S.o.pln("20");   X
}
X
C.E:  Constant Expression required.
```

Suppose

```
final int y=20;

Case y:
    S.o.pln("20");
```

→ If we declare y as final then we wont to get any Compiletime Error

→ Expressions are allowed for both Switch arguement & Case label but Case label should be Constant Expression

ex:-

```
int x=10;
Switch (x+1)
{
 Case 10:
        S.o.pln("10");      ✓
 Case 10+20:
        S.o.pln("10+20");
}
```

→ duplicate Case labels are not allowed.

ex:-

```
int x=10;
Switch(x)
{
  Case 97:
          S.o.pln("97");
  Case 98:
          S.o.pln("98");
  Case 99:  S.o.pln("99");
  Case 'a':
          S.o.pln("a");      X
}
```

C.E:- duplicate Case label

Summary:-

Case label

1. It should be Compile time Constant.
2. Expressions also allowed but should be Constant Expression
3. Value should be with the range of Switch argument type
4. Duplicates are not allowed.

fall-through inside switch :-

→ with in the Switch Statement if any Case is matched from That Case onwards all Statements will be executed until break Statement or End of the Switch. this is Called fall-through in inside Switch.

Ex:-

```
Switch (x)
{
  Case 0:
        S.o.pln ("0");

  Case 1:
        S.o.pln ("1");
        break;

  Case 2 :
        S.o.pln ("2");

  default:
        S.o.pln ("def");
}
```

o/p:-

if x=0 :-	if x=1 :-	if x=2	if x=3
0 1	1	2 def	def

→ fall-through inside Switch is useful to define Some Common action for Several Cases.

Eg! -

```
Switch (x)
{
  Case 3:
  Case 4:
  Case 5:
          S.o.pln ("Summer");
          break;
  Case 6:
  Case 7:
  Case 8:
  Case 9.
          S.o.pln("Rainny");
          break;
  Case 10:
  Case 11:
  Case 12:
  Case 1:
  Case 2:
          S.o.pln ("winter");
          break;
```

default case :-

→ We Can use default Case to define default action.

→ This case will be executed iff no other Case is matched

→ we can take default case any where within the Switch but it is Convension to take as Last case.

Eg! -

```
Switch (x)
{
  default: S.o.pln ("def");
  Case 0:  S.o.pln ("0");
           break;
  Case 1:  S.o.pln ("1");
  } Case 2: S.o.pln ("2");
```

$x=0$ → 0

$x=1$ → 1, 2

$x=2$ → 2

$x=3$ → def, 0

(b) Iterative Statements :-

(1) while :-

→ if we don't know the no. of iterations in advance then the best Suitable loop is while loop.

Ex:-

①
```
while (rs.next())
{
  --
}
```
↳ Result Set

②
```
while (itr.hasNext())
{
  --
  --
}
```
↳ Iterator

③
```
while (e.hasMoreElements())
{
  --
  --
}
```
→ enumeration

Syntax :-

```
while (b)
{
  Action
}
```
b → boolean type

→ The arguement to the while loop should be boolean type. if we are using any other type we will get Compiletime Error.

Ex:-
```
while (1)
{
  S.o.pln("Hello");
}
```
C.E :- incompatible types
found : int
required : boolean

→ Curlly brases are optional and without Curlly brases we Can take only one Statement which should not be declarative Statement.

Ex:①

```
while (true)
    S.o.pln("Hello");
```

✓

```
while(true);
```

✓

```
while (true)
int x=10;
```

✗

```
while (true)
{
 int x=10;
}
```

✓

Ex②:-

①

```
while (true)
{
 S.o.pln("Hello");
}
S.o.pln("Hi");
```

✗

C.E:- unreachable Statement

②

```
while (false)
{
 S.o.pln("Hello");
}
S.o.pln("Hi");
```

✗

C.E:- unreachable Statement

③

```
int a=10, b=20;
 while(a<b)
 {
 S.o.pln("Hello");
 }
 S.o.pln("Hi");
}
```

✓

o/p:- Hello
Hello
Hello
⋮

④

```
final int a=10, b=20;
 while (a<b)  → True
 {
 S.o.pln("Hello");
 }
 S.o.pln("Hi");
```

✗

Unreachable Statement

⑤ do-while :-

→ if we want to execute loop body atleast once then we should go for do-while loop.

Syn:-

```
do
{
  Action
} while(b);  ⟶ should be boolean type
          ;  ⟶ mandatory
```

→ Curlly brases are optional & without having Curlly brases we can take only one Statement b/w do & while which should not be declarative statement.

Ex:- ①

```
do
  S.o.pln("Hello");
while(true);
```

✓

②

```
do ;      ← is a valid java statement
while(true);
```

✓

③

```
do
int x=10;
while(true);
```

✗

④

```
do
{
int x=10;
}while(true);
```

✓

⑤

```
do
while(true);   ⟶ Compulsary one statement declare (or) take ';'
```

✗ C.E:-

⑥

```
do while(true)
  S.o.pln("Hello");
while(false);
```

✓

(or)

```
do
  while(true)
    S.o.pln("Hello");
while(false);
```

O/P:- Hello
Hello
;
;

note:-

";" is a valid java Statement

Eg:- ①

```
do
{
  S.o.pln("Hello");
}
while(true);
✗ S.o.pln("Hi");
C.E! unreachable Statement
```

②

```
do
{
  S.o.pln("Hello");
}
while(false);
S.o.pln("Hi");

o/p:- Hello
      Hi
```

③

```
int a=10, b=20;
do
{
  S.o.pln("Hello");
}
while(a<b);
S.o.pln("Hi");

o/p:- Hello
      Hello
      ⋮
```

④

```
int a=10, b=20;
do
{
  S.o.pln("Hello");
}
while(a>b);
S.o.pln("Hi");
✓
o/p: Hello
     Hi
```

⑤

```
final int a=10,b=20;
do
{
  S.o.pln("Hello");
}
while(a<b);
✗ S.o.pln("Hi");

C.E!- unreachable
        Statement
```

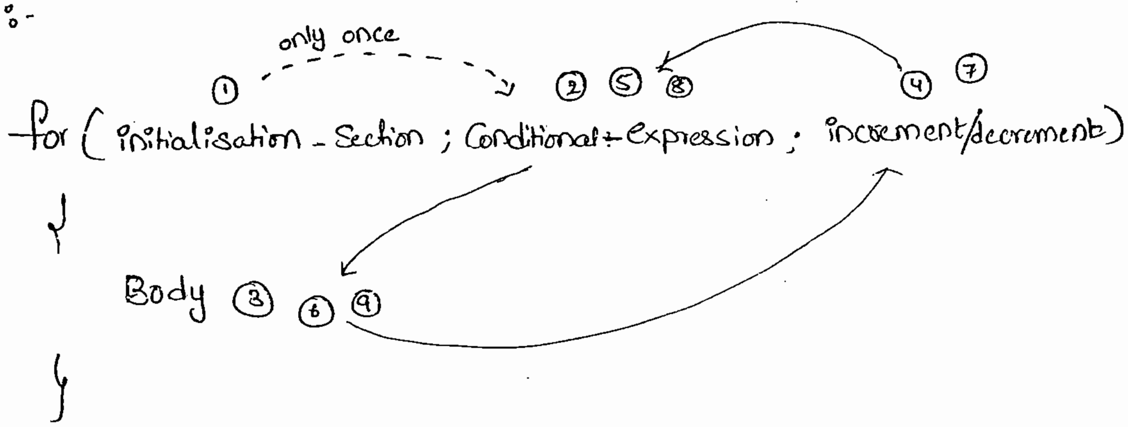
⑥

```
final int a=10,b=20;
do
{
  S.o.pln("Hello");
}
while(a>b);
S.o.pln("Hi");

o/p:- Hello
      Hi        ✓
```

for() :-

→ This is the most Commonly used loop

Syntax:-



```
for ( initialisation - Section ; Conditional-expression ; increment/decrement)
{
        Body
}
```

→ Curly braces are optional & without Curly braces we can take only one statement which should not be declarative statement.

(a) initialization-Section :-

→ This will be executed only once.

→ usually we are ~~perf~~ declaring and performing initialization for the variables in this Section.

→ Here we can declare multiple variables of the same type but different datatype variables we can't declare.

Ex:- ① int i=0, j=0; ✓

② int i=0 , byte b=0; ✗

③ int i=0, int j=0; ✗

→ in the initialization Section we can take any valid java statement including S.o.p() also

Ex:-

```
int i=0;
for (System.out.print("Hello U R Sleeping"); i<3 ; i++)
{
    S.o.pln(" No Boss U only sleeping");
}
```

O/P:-

```
Hello UR Sleeping
 No Boss  U only  Sleeping
 No Boss  U only  Sleeping
 No Boss  U only  Sleeping
```

Conditional Expression:-

→ Here, we can take any java Expression but the result should be boolean type.

→ It is optional and if we are not specifying then Compiler will always places "True".

Increment & decrement Section :-

→ we can take any valid java Statement including S.o.p() also.

Ex:-

```
int i =0;
for( S.o.pln("Hello"); i<3 ; S.o.pln ("Hi"))
{
    ~~S.o.pln(~~ i++;
}
```

O/P:-

```
Hello
Hi
Hi
```

→ All 3 parts of for loop are independent of each other.

→ All 3 parts of for loop are optional

ex!- `for ( ; ; );` T → Statement So, it is True.

⇒ represent infinite loop

Note:-

; is a Valid Java Statement

ex!-

```
for(int i=0; true; i++)   X
{
  S.o.pln("Hello");
}
S.o.pln("Hi");   X
```

C.E!- unreachable

```
for(int i=0; false; i++)   X
{
  S.o.pln("Hello");
}
S.o.pln("Hi");
```

C.E!- unreachable

```
for(int i=0; ; i++)   X
{
  S.o.pln("Hello");
}
S.o.pln("Hi");   X
```

C.E: unreachable

```
int a=10, b=20;
for(int i=0; a<b; i++)
{
  S.o.pln("Hello");
}
S.o.pln("Hi");
```

O/P!- Hello ✓
Hello
⋮

```
final int a=10; b=20;
for(int i=0; a<b; i++)
              True
{
  S.o.pln("Hello");
}
S.o.pln("Hi");   X
```

O/P:- C.E!- unreachable Statement.

for-each() Loop:- (Enhanced for loop):.

→ Introduced in 1.5v. This

→ This is the most Convenient loop to retrieve the elements of Arrays & Collections

Ex:- ① print elements of Single dimensional Array by using General & enchanced for loops

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

for-loop

```
for(int i=0; i<a.length; i++)
{
  S.o.pln(a[i]);        10
                        20
}                       30
                        40
                        50
```

for-each

```
for (int x : a)
{
  S.o.pln(x);           10
                        20
}                       30
                        40
                        50
```

② print the elements of 2D-int Array by using General & for-each loop

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

for-loop

```
for(int i=0; i<a.length; i++)
{
  for (int j=0; j<a[i].length; j++)
  {
    S.o.pln(a[i][j]);       10
                            20
} }
```

for-each

```
for (int[] x : a)
{
  for (int y: x)
  {
    S.o.pln(y);         10
                        20
                        30
                        40
                        50
} }
```

→ Even though for-each loop is more Conveient to Use, but it has the following limitations.

(i) It is not a General purpose loop –

(ii) It is applicable only for Arrays & Collections

(ii) By using for-each loop we should retrive all values of Arrays & Collections and Can't be used to retrieved a particular Set of Values.

(C) Transfer statements :-

(1) break :-

→ We Can use break Statement in the following Cases

(1) within the Swith to stop fall through

(2) inside loops to break the loop execution based on Some Condition

(3) inside labeled blocks to break that block execution based on some Condition.

Ep:-

```
Switch (b)
{
   !
   break;
   !
}
```

```
for(int i=0; i<10; i++)
{
 if (i==5)
 {
   break;
 S.o.pln (i);
 :
```

```
Class Test
{
 p.s.v.m ( —— )
 {
   int i=10;
   l1:
   {
    S.o.pln(" Hello");
   if(i = =10)
    break l1;
   S.o.pln(" Hi");
   }
   S.o.pln(" End");
 }}
```

op:
Hello
End

→ if we are useing break Statement Any where else we will get Compiletime Error

Ex!-

```
Class Test
{
  p.s.v.m( ——)
  {
   int x=10;
   if(x==10)
      break;
   S.o.pln("Hello");
  }
}
```

C.E break outside Switch or loop.

Continue Statement:-

→ we Can use Continue Statement to Skip Current iteration and Continue for the next iteration inside loops

Ex!.

```
for(int i=0 ; i<=10 ; i++)
{
  if(i%2==0)
     Continue;
  S.o.pln(i);
}
```

```
1
3
5
7
9
```

→ If we are using Continue outside of loops we will get Compiletime Error.

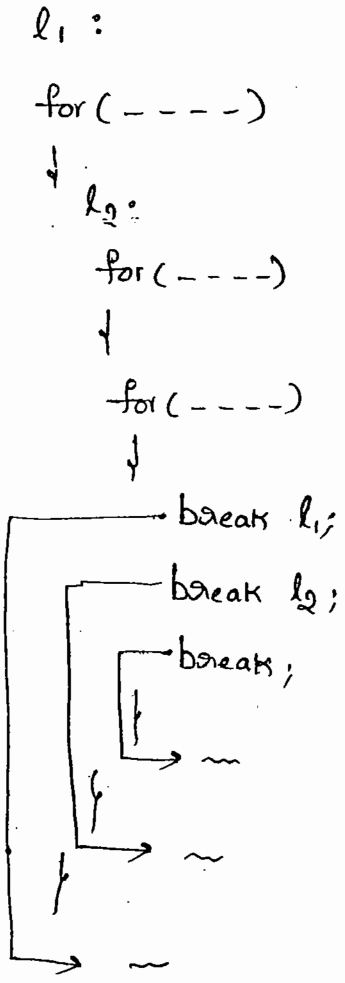
Ex:-

```
int x=10;
if(x==10)
    continue;   → ✗   C.E:- Continue outside of loop
S.o.pln("Hello");
```

labeled break & Continue Statements:-

→ In the Case of nested loops to break and Continue a particular loop we should go for labeled break & Continue statements.

Ex:-



```
l1 :
for(----)
{
   l2 :
   for(----)
   {
     for(----)
     {
       break l1;
       break l2;
       break;
     }
     ~
   }
   ~
}
~
```

ex2:-

```
l1:
for(int i=0; i<3; i++)
{
  for(int j=0; j<=3; j++)
  {
    if(i==j)
       break;
    S.o.pln(i+"-----"+j);
  }
}
```

break:-

```
1 ...... 0
2 ...... 0
2 ------ 1
```

break l1:-

No output

Continue:-

```
0 ---- 1     2 --- 0
0 --- 2      2 --- 1
1 ---- 0
1 ---- 2
```

Continue l1:-

```
1 .... 0
2 - -- 0
2 ~~ 1
```

do-while Vs Continue :- (Very hot Combination)

Ex!-

```
int x=0;
do
{
  x++;
  S.o.pln(x);
  if(++x <5)
    Continue;
    x++;
  S.o.pln(x);
} while (++x<10);
```

1
4
6
8
10

x=1

(i)

0
1<5

x = ~~0~~ 1

~~1~~ ~~2~~<5 ~~3~~<10 1
~~4~~ 5<5 4
x++ 6
~~6~~
~~7~~<10
~~8~~ 8
~~9~~
~~10~~
1

Imp Note!-

→ Compiler will check for unreachable Statements only in the Case of loops but not in 'if – else'.

Ex!. ①

```
if (true)
{
S.o.pln(" Hello");
}
else
{
S.o.pln("Hi");
}
```

o/p!- Hello

②

```
while(true)
{
  S.o.pln('Hello');
}
S.o.pln("Hi");
```

o/p!- C.E :-
unreachable Statement

Declarations & Access Modifiers

1. Java Source file Structure (1-9)   Packages - 8
2. Class modifiers (10-14)
3. member modifiers (15-23)

* 4. Interfaces. (24-31)

Java Source file Structure :-

→ A Java program Can Contain any no. of classes but atmost one class Can be declared as the public. if there is a public class the name of the program & name of public class must be matched otherwise we will get CompiletimeError.

→ If there is no public class then we Can use any name as Java Source file name, There are no restrictions.

Ex :-

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

Save: Sai.java (or) ✓
R.java (or) ✓
D.java. ✓

Case(1):-

If there is no public class then we can use any name as java source file name.

Ex:- A.java ✓
B.java ✓
C.java ✓
Durga.java ✓

Case2:->

If class B declared as public & the program name is A.java then we will get CompiletimeError saying,

" Class B is public should be declared in a file named B.java"

Case 3:-

If we declare Both A & B classes as public & name of the program is B.java then we will get CompiletimeError saying.

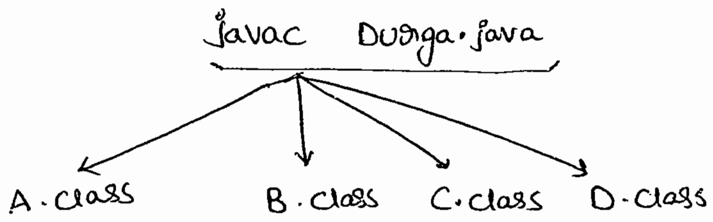
" Class A is public should be declared in a file named A.java".

Ex:-

```
class A
{
  P.S.V.m(String[] args)
  {
    S.o.pln(" A class main method");
  }
}
Class B
{
  p.S.v.m(String[] args)
  {
    S.o.pln("B class main method");
  } }
```

```
Class C
{
  P.S.v.m(String[] args)
  {
    S.o.pln("C class main method");
  }
}
Class D
{
}
```

Save ⇒ Durga.java



① java A ↵

A class main method

② java B ↵

B class main method

③ java C ↵

C class main method

④ java D ↵

R.E:- NoSuchMethodError: main

⑤ java Durga ↵

R.E:- NoClassDefFoundError: Durga

Note 1..

→ It is highly recommended to take only one class per Source file & name of the file and that class name must be matched. This approach improves readability of the Code.

Import Statement :-

```
Class  Test
{
 P.S.V.m(String[] args)
 {
  Arraylist  l = new Arraylist();     // Arraylist()
 }
}
```

C.E! — symbol: method Arraylist

C.E!- Cannot find Symbol
Symbol: class Arraylist
Location: class Test

→ We can resolve this problem by using fully Qualified name java.util.

→ The problem with usage of fully Qualified name every time increases length of the Code & reduces readability.

→ We can resolve this problem by using import Statement

```
import java.util.Arraylist;
Class Test {
 P.S.V.m(String[] args)
 {
  AL l = new AL();
 }}
```

→ Where ever we are using import statement it is not required to use fully Qualifiedname hence it reduces improves readability & reduces Length of the Code.

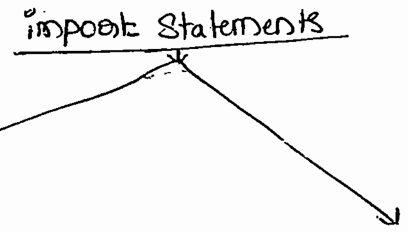
Case(1):-

Types of import Statements:-

↳ There are 2 types of import Statements

(1) Explicit class import

(2) Implicit class import



Explicit class import :-

Ex!. import java.util.AL;

→ This type of import is highly recommended to use. because it improves readability of the Code.

→ Best Suitable for Hitech City Where readability is importent

Implicit class import :-

Ex!. import java.util.*;

→ It is never recommended to use this type of import because it redues readability of the Code.

→ Best Suitable for Ameerpet where typeing is important.

Case 2!. difference b/w #include & import Statement :-

→ In c language #include all the Specified header files will be loaded at the time of include Statement only irrespective of wheather we are using those header files are not. Hence this is "Static loading".

→ But in the case of Java language import Statement no .class file will loaded at the time of import statement, in the next lines of code when ever we are using a class at that time only the Corresponding .class file will be loaded. this type of loading is called dynamic loading or load on demand or load on fly.

Case 3!

which of the following import Statements are valid?

✗ ① import java.util;

✗ ② import java.util.AL.*;

✓ ③ import java.util.*;

✓ ④ import java.util.AL;

Case 4!.

→ Consider the Code,

```
Class MyRemoteObject Extends Java.rmi.UniCast
                                    RemoteObject
{
   =
}
```

→ the Code Compiles fine Eventhough we are not using import Statement because we used fullyQualified Name.

Note!-

→ when ever using fully Qualified name it is not required to use import Statement. When ever we are using import Statement it is not required to use fully Qualified name.

~~Case5:~~
~~Exa~~mple :-

Date } available in both { util
List }   { sql

```
import java.util.*;
import java.sql.*;
class Test
{
 p.s.v.m(String[] args)
 {
   Date d = new Date();
 }
}
```

C.E :- " Reference to Date is ambiguouse .

Note :-
even in List Case also we will get the same ambiguity problem because it is available in both Util & Sql packages.

Case6 :-

```
import java.util.Date;
import java.sql.*;
Class Test
{
  p.s.v.m (String[] args)
  {
    Date d = new Date();
  }
}
```

order :.

✓ ① Explicit class import
✓ ② Classes present in Current working directory
③ implicit class import.

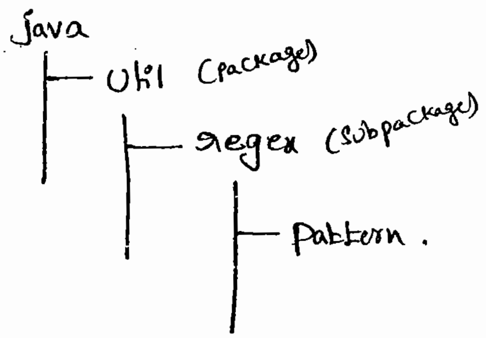
Conclusion :- While Resolveing Class names Compiler will always gives the precedence in the following order,

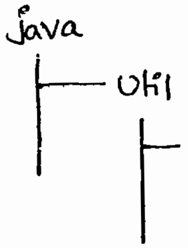
→ order see above

Case 7:-

→ When ever we are importing a package all classes & interfaces present in that package are available, but not Subpackage classes.

Ex:-





→ To use pattern class which of the following import is required

✗ ① import java.*;

✗ ② import java.util.*;

✓ ③ import java.util.regex.*;

✓ ④ import java.util.regex.pattern;

Case(8):-

→ the following 2 packages are not required to import because all classes & interfaces present in these 2 packages are available by default to every java program.

① java.lang package.

② ~~Java~~ default package(current working directory).

Case 9:-

→ import Statement is totally compiletime issue if no.of imports increases then compiletime will be increased automatically, but there is no effect on Execution time.

Static import :-

→ This Concept introduced in 1.5 Version.

→ According to SUN Static import improves readability of the Code, But according to World wide programming Experts (Like us) Static imports reduces the readability of the Code & creates Confusion, it is not recommended to use Static import if there is no Specific requirement

→ Usually we can access Static members by using class names, but whenever we are using Static import, it is not required to use Class name and we can access Static members directly.

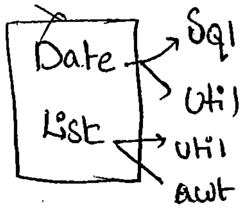
ex:-

Without Static import

```
class Test
{
 p.s.v.m(String[] args)
 {
  S.o.pln(Math.sqrt(4));
  S.o.pln(Math.random());
  S.o.pln(Math.max(10,20));
 }
}
```

With Static import

```
import static java.lang.math.sqrt;
import static java.lang.math.*;
class Test
{
 p.s.v.m(String[] args)
 {
  S.o.pln(sqrt(4));
  S.o.pln(random());
  S.o.pln(max(10,20));
 }
}
```



* Explain about System.out.println():-

```
Class Test
{
p static String name = "xyz";
}
Test.name.length();
```

It is a class name

Static variable present in Test class of the type String

It is a method present in String class

```
Class System
{
Static PrintStream out;
}
System.out.println();
```

It is a Class present in java.lang package

It is a Static variable of type printStream present in System Class

It is a method present in PrintStream class.

Explanation:

→ Out is a static variable present in System Class hence we can access by using classname.

→ But when ever we are using static import it is not required to use class name we can access out variable directly.

```
import static java.lang.System.out;
Class Test
{
 p.s.v.m(String[] args)
 {
  out.println("Hello");   Hello
  out.println("Hi");   Hi
 }
}
```

-ve perspective (Ambiguity):-

Sol:
```
import static java.lang.Integer.*;
import static java.lang.Byte.*;

 Class Test
 {
  p.s.v.m(String[] args)
  {
    S.o.pln(MAX_VALUE);
  }
 }
```

C.E:- reference to MAX_VALUE in ambiguity

Note:-

Two classes Contains a variable or method with Same name is Very Common Hence ambiguity problem is also Very Common in Static import.

Ex:-

→ While resolveing static members Compiler will always gives the precedence in the following order.

1. Current class Static members
2. Explicit Static import
3. implicit Static import.

Ex:-

```
import Static java.lang. Integer. MAX-VALUE;  ——→ ②
import Static java.lang. Byte .*;  ——→ ③
Class Test
{
  Static int MAX-VALUE =999;  ——→ ①
  P.S.v.m(String[] args)
  {
    S.o.pln (MAX-VALUE);
  }
}
```

→ If we are Commenting Line① Then Explicit Static import will get priority Hence we will get Integer class MAX-VALUE is o/p 2147483647.

→ If we are Commenting Lines① & ② Then Byte Class MAX-VALUE will be Considered & we will get 127 as o/p.

(-ve point):-

→ Strictlly Speaking usage of Class Name to access Static variables & methods improves readability of The Code. Hence it is not recommended to use Static imports.

Q) which of the following import Statements are valid.

✗ ① import java.lang.math.*; (we should not use * after the class).

✗ ② import java.lang.math.Sqrt.*; (we should not use * after the method).

✗ ③ import Static java.lang.math;

✓ ④ import java.lang.math;

✓ ⑤ import Static java.lang.math.*;

✗ ⑥ import Static java.lang.math.sqrt(); ⟶ problems

✓ ⑦ import Static java.lang.math.sqrt;

Normal import Vs Static import:-

→ we Can use normal import to import classes & interfaces of a package. when ever we are using general import it is not required to use fully Qualified Name & we Can use short names directly.

→ we Can use Static import to import Static variables & methods of a class. when ever we are using Static import then it is not required to class name to access Static member we Can acess directly.

Packages

9346 classes are their in Java according to 1.6v

Package:-

→ It is an Encapsulation mechanism to group related classes and interfaces into a single module. The main purposes of packages are

① To resolve naming conflicts.

② To provide Security to the classes & interfaces. So that out side person can't access directly

③ It improves modularity of the application.

→ There is one universally accepted Convension to name packages i.e to use internet domain name in reverse.

$$\underbrace{\text{Com.icicibank}}_{\text{domain name in reverse}}.\underbrace{\text{loan}}_{\text{module name}}.\underbrace{\text{housingloan}}_{\text{Submodule name}}.\underbrace{\text{Account}}_{\text{Class name.}}$$

Ex:-

```
Package Com.durgajobs.itjobs;
Public class HydJobs
{
 P.S.v.m(String[] args)
 {
  S.o.pln(" Getting Jobs is very easy");
 }
}
```

① javac HydJobs.java

→ The generated class file will be placed in Current working directory

```
CWD
 |
 |--> HydJobs.class
```

② javac -d . HydJobs.java

→ destination to place generated class files

current working directory

→ generated class file will be placed into Corresponding package Structure.

```
CWD
 |-com
    |
    |----durgajobs
            |
            |---itjobs
                  |
                  |-- HydJobs.class
```

→ If the specified package structure is not already available then this Command itself will Create that package Structure.

→ As the destination we Can use any valid directory

Ex1. javac -d c: HydJobs.java

```
c:
 |-com
    |
    |---durgajobs
            |
            |---itjobs
                  |
                  |----HydJobs.class
```

→ If the Specified destination is not already available then we will get Compile time Error

Ex: javac -d Z: HydJobs.java

→ If Z: is not already available then we will get Compiletime Error.

Run

→ Java com.durgajobs.itjobs.HydJobs ↵

o/p:- Getting Job is Very easy.

Conclusions:-

① → In Any Java program there should be only Atmost 1 packge Statement. If we are taking morethan one package Statement we will get Compiletime Error.

Ex:-

```
✓ package pack1;
→ package pack2; ←
  class A
  {
  }
                C.E:- class, interface or enum expected.
```

② In Any Java program the first non Comment Statement should be package Statement (if it is available).

Ex:-

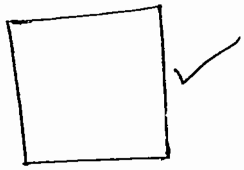
```
✓ import java.util.*;
→ package pack1;
  class A
  {
  }
C.E:- class, interface or enum Expected.
```

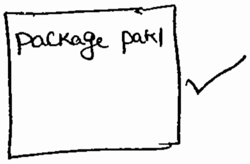
→ The proper Structure of a Java Source file is

Order is important ↓

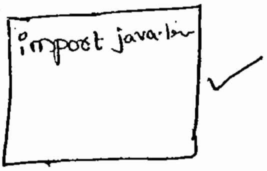
Package Statement;	→ Atmost one
Import Statements;	→ Any number
class/interface/enum declarations	→ Any number

→ The following are Valid Java programs.

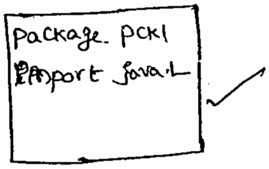
①

Test.java

②


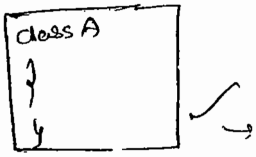
Test.java

③


Test.java

④


Test.java

⑤


→ in this prog only creates .class file.

Test.java

→ An Empty Source file is a Valid Java program.

** Class modifiers **

→ when ever ~~yo~~ we are writing our own java class Compulsary we have to provide Some information about our class to the JVM Like,

(1) Wheather our class can be accessable from any where or not

(2) wheather child class Creation is possible for our class or not.

(3) wheather instanciation is possible or not e.t.c.

→ we Can Specify this information by declaring with appropriate modifier.

→ The only applicable modifiers for top-level Classes are

1) public
2) <default>
3) final
4) abstract
5) Strictfp.

→ If we are using any other modifier we will get Compiletime Error. Saying "modifier xxxxxxx not allowed here".

Ex:-

```
Private class Test
{
  p.s.v.m(____)
  {
   int x=0;
   for(int y=0; y<3; y++)
   {
    x=x+y;
   }
   S.o.pln(x);
}}
```

C.E!- modifier private not allowed here

→ But for the Inner classes the following modifiers are allowed

(1) public
(2) <default>
(3) final
(4) abstract
(5) strictfp
(6) private
(7) protected
(8) Static.

```
private class A
{
}
Static class B
{
}
p.s.v.m( —)
{
  S.o.pln("Hi");
}
```

because the main class is not declare

28/04/11

access Specifiers Vs access modifiers :-

→ In old languages like C & C++ Public, private, protected & default are Considered as access Specifiers .. & all the remaining like final, Static are Considered as access modifiers.

→ But in Java there is no Such type of division all are Considered as access modifiers.

Public classes :-

→ If a Class declared as the public then we can access that class from any where.

Ex:-

```
package pack1;
Public class A
{
 Public void m1()
 {
  S.o.pln("Hello");
 }
}
```

Javac -d . A.java

```
CWD
 ├── pack1
       ├── A.class
```

```
Package pack2;
import pack1.A;
Class B
{
 P.S.V.m(String[] args)
 {
     A a = new A();
     a.m1();
 }
}
```

Compl. javac -d . B.java ↵

Run: java pack2.B ↵

→ If we are not declaring Class A as public, then we will get Compile time Error while Compaling B class, Saying "pack1.A is not public in pack1; Cant be accessed from outside Package"

default classes :-

→ If a class declared as default then we Can access that class only With in that Current package. i.e from outside of the package we Can't access.

final modifier :-

→ final is the modifier applicable for classes, methods & variables.

→ If a method declared as the final then we are not allowed to ~~to~~ override that method in the child class.

ex:-

```
        Class P
        {
          Public void property()
          {
           S.o.pln(" money + Gold + Land");
          }
          Public final void marry()
          {
           S.o.pln(" Subba laxmi ").
          }
C.E     }
        Class C extends P
        {
          public void marry()
          {
           S.o.pln(" Kajal | 3sba | atara ");
          }
        }
```

C.E!- marry() in C cannot override marry() in P ; overridden method is final.

→ If a class declared as the final then we can't Create child class

ex:-

```
final Class P          class C extends P
{                      {
}                      }   X
```

Ex:-

```
final class P
{
}
class C extends P
{
}
```

C.E:- Can't inherit from final P.

↳ Every method present inside a final class is always final bydefault. but Every Variable present in final class need not be final.

→ The main Advantage of final keyword is we can acheive Security as no one is allowed to change our implementation.

→ But the main disAdvantage of final Keyword is we are missing Key benefits of Oop's Inheritance & polymorphism (overriding). Hence, if there is no Specific requirement never recommended to use final keyword.

**abstract modifier:-

→ abstract is the modifier applicable for classes & methods but not for variables.

abstract method:-

→ Eventhough we dont ~~about,~~ Know about Implementation Still we Can declare a method with abstract modifier. i.e abstract methods Can have only declaration but not implementation. Hence, Every abstract method declaration should Compulsary ends with ";".

Ex:-

```
X  1) public abstract void m1(){ }

✓  2) public abstract void m2();
```

→ Child classes are responsible to provide implementation for parent class abstract methods.

Ex:-

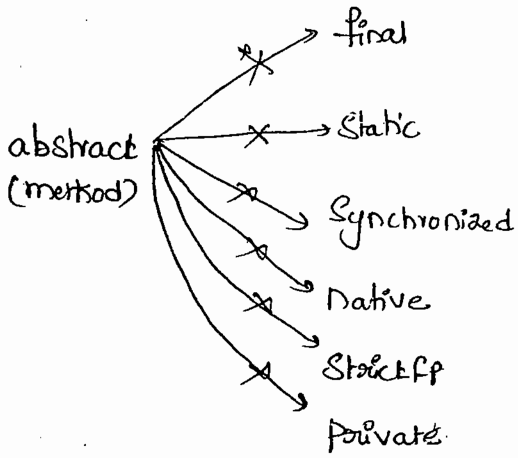
```
        abstract class Vechicle
        {
          public abstract int getNoofwheels();
        }

        Class Bus extends Vehicle
        {
          public int getNoOfWheeels()
          {
            return 6;
          }
        }
        Class Auto extends Vehicle
        {
          public int getNoOfWheels()
          {
            return 3;
          }
        }
```

→ By declaring abstract methods in parent class we can define Guidelines to the child classes which describes the methods those are to be Compulsary implemented by child class.

29/04/11

→ abstract modifier never talks about implementation, if any modifier talks about implementation then it is always illegal Combination with abstract.

→ The following are various illegal Combinations of modifiers for methods



abstract class :-

→ for any java class if we don't want instanciation then we have to declare that class as abstract. i.e, for abstract classes instanciation (creation of object) is not possible.

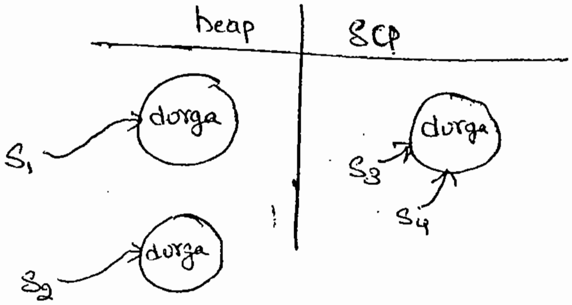
Ex:-

```
abstract class Test
{

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

C.E:- Test is abstract ; Cannot be instantiated
Test t =new Test();
^

Ex:-

```
String S1 = new String("durga");
String S2 = new String("durga");
String S3 = "durga";
String S4 = "durga";
```



ohm SaiRam prabu.

1) notify() & notifyall()

2) Collection & Collections

3) equals() & ==

4) Comparable & Comparator

5) String & StringBuffer

6) StringBuffer & String Builder

7) throw & throws

8) throws & thrown

9) HashMap & Hashtable

10) enum, Enum, Enumeration

11) final, finally, finalizer

durga sir.

✓ 1) Language fundamentals

✓ 2) Operators & Assignments

3) flow-Control

4) declaration & Access modifier

5) oops Concepts

✓ 6) Exception Handling

7) multi threading

✓ 8) Inner classes

9) java.lang package

✓ 10) java.io.package

11) Serialization

✓ 12) java.util.package (Collection frame work)

13) Generics

14) Regular Expressions

✓ 15) G.c

✓ 16) Assertions (1.4)

17) I18N

18) enum

19) developement

Dell

1) chaitanyajobs@gmail.com

Satish4dworld@ "

29444524

Chait

Chaithanya-Anumanchi@dell.com.

ON DurgajobsInfo 9870807070

abstract class Vs abstract method :-

→ If a class Contains atleast one abstract method then Compulsary That class should be declared as abstract otherwise we will get Compiletime Error. because, the implementation is not Complete & hence we Can't Create an object.

→ Eventhough Cas class doesnot Contain any abstract method still we Can declare the class as abstract. i.e, abstract class Can Contain Zero "0 no.of abstract method.

Ex:- HTTPServelet, this class doesn't Contain any abstract method but still it is declared as abstract.

Ex:-

①
```
Class Test
{
  public void m1();
}
```
C.E:- missing method body, or declare abstract

②
```
Class Test
{
  public abstract void m1()
  {
  }
}
```
C.E:- abstract methods Can't have a body

③
```
Class Test
{
  public abstract void m1();
}
```
C.E:- Test is not abstract and doesn't override abstract method m1() in Test.

Ex-4!-

```
abstract class Test
{
  public abstract void m1();
  public abstract void m2();
}
class SubTest extends Test
{
   public abstract void m1() { }
}
```

CE!- SubTest is not abstract and does not override abstract method m2() in Test

→ we can handle these compiletime Errors either by declaring class SubTest as abstract or by providing implementation for m2().

Note!-

→ The usage of abstract methods, abstract class & interfaces are recommended & it is always good programming practice.

abstract Vs final :-

→ abstract methods we have to override in child classes to provide implementation. where as final methods can't be overridden. Hence, abstract final combination is illegal combination for methods.

→ for abstract classes we should create child classes to provide proper implementation but for final classes we can't create child class. Hence abstract final combination is illegal for classes.

→ final class Can't have abstract methods where as abstract class Can Contain final methods.

```
final class A
{
   abstract void m1();
}
```

X

```
abstract class A
{
   public final void m1()
   {
   } =
}
```

✓

Strictfp (all Lower case) modifier :- (strictfloatingpoint)

→ Strictfp is the modifier applicable for methods & classes but not for variables.

→ if a method declared as strictfp all floatingpoint Caluclations in that method has to follow IEEE 754 Standard So, that we will get Platform independent results.

→ Strictfp method always talks about implementation where as abstract method never talks about implementation. Hence strictfp-abstract method Combination is illegal Combination for methods.

→ If a class declared as strictfp then every Concreate method in that class has to follow IEEE 754 Standard So, that we will get Platform independent results.

→ abstract – strictfp Combination is legal for classes but illegal for methods

ex:-

```
abstract strictfp class Test
{
}
```

✓

public abstract strictfp void m1(); X (invalid).

Member (variables & methods) modifiers :-

① public members :-

→ If we declare a member as public then we can access that member from anywhere but corresponding class should be visible (public) i.e, Before checking member visibility we have to check class visibility.

ex:-

```
package pack1;

→ class A
  {
    public void m1()
    {
      S.o.pln("Hi");
    }
  }
```

```
package pack2;
import pack1.A;
class B
{
  p.s.v.m(——)
  {
     A a = new A();
     a.m1();
  }
}                X
```

→ Eventhough m1() method is public, we can't access m1() from outside of pack1 because the corresponding class A is not declared as public. if both are public then only we can access.

② default members:-

→ If a member declared as the default, then we can access that member only with in the current package & we can't access from outside of the package. Hence, default access is also known as package level access.

③ private members :-

→ If a member declared as private then we can access that member only with in the current class.

→ abstract methods should be visible in child classes to provide implementation where as private methods are not visible in child classes. Hence private-abstract combination is illegal for methods.

④ protected members : (The most misunderstood modifier in java) :-

→ If a member declared as protected then we can access that member with in the current package any where but outside package only in child classes.

$$\boxed{\text{protected} \equiv \langle \text{default} \rangle + \text{kids of}}$$ an another package (only child reference).

*→ with in the current package we can access protected members either by parent reference or by child reference.

→ But from outside package we can access protected members only by using child reference. if we are trying to use parent reference we will get C.E

Eg:-

```
package pack1;
public class A
{
  protected void m1()
   {
    S.o.pln("The most misunderstood modifier in Java");
   }
}
class B extends A
{
  p.s.v.m(——)
  {
```

```
✓ A a = new A()
  ✓ | a.m1();
✓ B b = new B()
  ✓ | b.m1();
✓ A a1 = new B()
  ✓ | a1.m1();
```

```
Package pack2;
 import pack1.A;
 public class C extends A
 {
  p.s.v.m(___)
  {
      A a = new A()
  ✗   a.m1();
      C c = new C()
  ✓   c.m1();
      A a1 = new C()
  ✗   a1.m1();
  }
 }
```

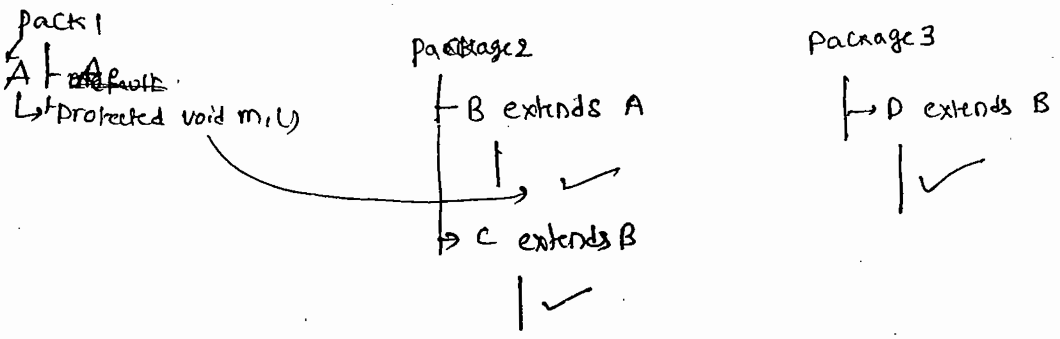
→ The most restricted modifier is 'private'

→ The most accesible modifier is 'public'

→ private < default < protected < public

→ The recommended modifier for variables is private

→ The recommended modifier for methods is public



→ The most restricted

* private < default < protected < public

Visibility	private	<default>	protected	public
① with in the Same Class	✓	✓	✓	✓
② from child class of Same Package	✗	✓	✓	✓
③ from non-child class of Same package	✗	✓	✓	✓
④ from child class of outside Package.	✗	✗	✓ (But we should use only child class reference	✓
⑤ from non-child class of outside package.	✗	✗	✗	✓

"final" variables :-

→ In General for instance & static variables it is not required to perform initialization Explicitly JVM will always provide default Values.

→ But for the local variables JVM won't to provide any default Values Compulsory we should provide initialization before using that Variable.

"final instance variables" :-

→ for the normal instance variables it is not required to perform initialization Explicitly JVM will provide default values.

→ If the instance variable declared as the final then Compulsory we should perform initialization wheather we are using or not otherwise we will get Compiletime Error

ex:-

```
Class Test
{
    int x;
}   ✓
```

```
Class Test
{
    final int x;
}   X
```

C.E:- Variable x might ~~have~~ not have been initialized.

Rule:-

(**) for the final instance variables we should perform initialization before Constructor Complition.

→ i.e, The following are various places for this,

① At the time of declaration

ex:-

```
class Test
{
    final int x=10;
}
```
✓

② inside instance Block.

ex:-

```
Class Test
{
    final int x;
    {
      x=10;
    }// instance Block
}
```
✓

③ inside Constructor.

ex:-

```
Class Test
{
    final int x;
    Test()
    {
      x=10;
    }
}
```
✓

→ Other than these if we are perform Initialization any where else we will get Compiletime Error.

Ex:-

```
Class Test
{
   final int x;
    public void m1()
    {
        x=10;    X      C.E!- Cannot assign a value to
    }                         final Variable x.
}
```

final Static Variables:-

→ for the normal Static Variables it is not required to perform initialization Explicitly, JVM will always provide default values.

→ But for final static Variables we shoud perform initialization Explicitly Otherwise we will get C.E.

Ex:-

```
Class Test              |  Class Test
{                       |  {
  Static int x;         |   final static int x;
}                       |
            ✓           |  }  X
```

C.E!- Variable x might not have been initialized.

Rule:-

* for the final Static Variables we should perform initialization Before Class loading Complition.

* i.e, the following are Various places to perform this.

① At The time of declaration

Eg!-

✓

```
Class Test
{
    final static int x=10;
}
```

② Inside Static Block

Eg!-

✓

```
Class Test
{
    final static int x;
    Static
    {
      x=10;
    }
}
```

→ If we are performing initialization any where else we will get Compiletime Error.

```
Class Test
{
  final static int x;
   Public void m1()
   {
       x=10;     X
   }
}
```

C.E!- Can't assign a ~~variable~~ Value to final variable x.

iii) final Local Variables :-

→ for the local variables JVM won't to provide any default values Compulsary we should perform initialization before useing that variable.

Ex!- ①

```
Class Test
{
  public S void main( )
  {
    int x;
    S.o.pln("Hello");   ✓
  }
}
```

O/p!- Hello

②

```
Class Test
{
  public S void main( )
  {
    int x;
    S.o.pln (x);   X
  }
}
```

C.E!- variable x might not have been initialized.

④ → Eventhough Local variable declared as the final it is not required to Perform initialization if we are not using that variable.

Ex!

```
Class Test
{
  P. S. v.m( )
  {
    final int x;
    S.o.pln ("Hello Sai");   ✓
  }
}
```

O/p !- Hello Sai.

*** → The only applicable modifier for local variables is final. If we are using any other modifier we will get Compiletime Error.

ex:-

```
Class Test {
    p.s.v.m() {
        X public int x=10;
        X private int x=20;
          static int x=60; X
          protected int x=30; X
          final int x=40; ✓
```

→* formal parameters of a method Simply access as Local variables of That method. hence, a formal parameter Can be declared as final.

→ If we declare a formal parameter as final within the method we Can't change its value otherwise we will get Compiletime Error.

Ex:-

```
Class Test
{
  p.s v.m(----)
  {
      m1(10,20);
  }       └──→ Actual parameters

p.s.v.m1(final int x, int y)
{                      └→ formal parameters

   x=1000;  // Can't assign a value to final variable x.
   y=2000;

   S.o.pln(x+"----"+y);
 }
}
```

Static → class level
instance → object level

Static modifier:-

→ Static is the modifier applicable for variables & methods but not for Classes (but innerclass Can be declared as Static).

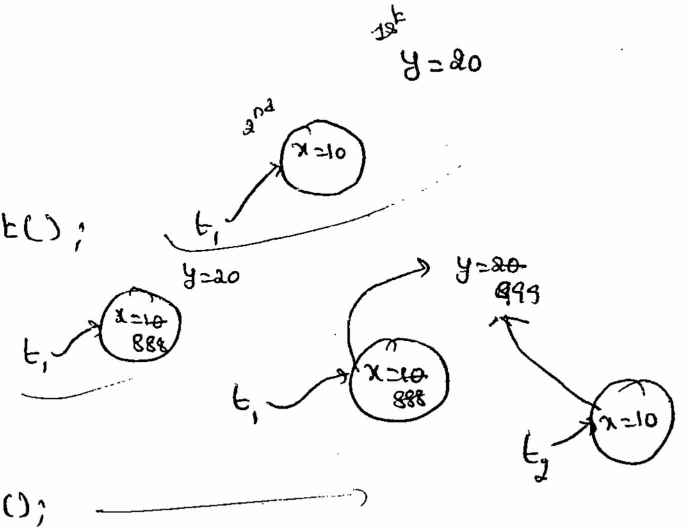
→ if the value of a variable is varied from Object to Object then we Should go for instance variable. In the Case of Instance variable for Every object a Seperate Copy will be Created.

→ If the value of a variable is Same for all objects then we should go for Static variables. In the Case of Static variable Only one Copy will be Created at class Level and share that Copy for every object of that class.

first Static variable is Created at the begining when class is Created. 19

Ex:-

```
Class Test
{
  int x=10;
  Static int y=20;
  P.S.v.m(——)
  {
   Test t1 =new Test();
    t1.x = 888; →
    t1.y = 999; →
   Test t2 =new Test();
   S.o.pln(t2.x+"---"+ t2.y);
  }          10              999
}
```



for every object a Seperate copy will be Created.

→ Static members Can be accessed from both instance & Static areas where as instance members Can be accessed only from instance area directly. i.e, from static area we Can't access instance members directly otherwise we will get Compiletime Error.

Q) Consider the following declarations

I. int x=10;

II. Static int x=10;

III. Public void m1()
{
  S.o.pln(x);
}

IV. Public Static void m1()
{
  S.o.pln(x);
}

→ which of the above we can take Simultaneously with in the Same class.

✓ A). I & III

✗ B). I & IV . CE :- non-Static variable x Cannot be accessed from static Context

✓ C) II & III

✓ D) II & IV

✗ E) I & II

✗ F) III & IV

→ for Static methods Compulsary implementation Should be available where as for abstract methods implementation should not be available Hence abstract-Static Combination is illegal for methods.

→ for Static methods overloading Concept is applicable Hence with in the Same class we can declare 2 main methods with different arguments

Ex:-

```
Class Test
{
  p. s. v.m(String[] args)
   {
     S.o.pln(" String[] ");
   }
 public static void main( int[] args)
 {
   S.o.pln(" int[] ");
 }
}
              O/P:- String[]
```

→ ~~But JVM also~~

→ But Jvm always Call String arguements main method only. The other main method we have to Call explicitly just Like a normal method Call.

→ Inheritance Concept is applicable for Static methods including main() method hence while executing child class if the child doesnot Contain main method then the parent Class main method will be execute

ex:-
```
Class p
{
  p.s.v.m(String[] args)
  {
     S.o.pln(" parent Class");
  }
}
Class c extends p
{
}
```

```
      Javac p.java
     /            \
P.class          C.class
```

o/p java p   o/p java c
Parent class   parent class.

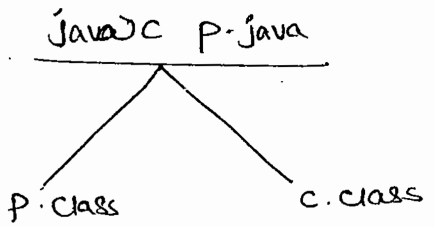
→ It Seems that overriding Concept is applicable for Static methods but it is not overriding, it is method hiding.

Ex:-
```
Class P
{
  P.s.v.m( ——)
```

Ex!-

```
Class P
{
 p.s.v.main(String[] args)
 {
  S.o.pln(" Parent classmain");
 }
}
Class C extends P
{
 p.s.v.main(String[] args)
 {
  S.o.pln(" child method");
 }
}
```

it is not overriding
it is method hiding



```
java P ↵
 Parent main

java C ↵
  Child main .
```

native modifier :-

→ native is the modifier applicable only for methods but not for variables and classes.

→ The native methods are implemented in some other languages like C & C++ hence native methods also known as "foreign methods".

→ The main objectives of native keyword are ① to improve performance of the System

① To improve performance of the System.

② To use already existing legacy non-Java code.

Pseudo Code :-

→ To use native keyword

ex:-

```
                     class Native
                     {
                        static
                        {
① Load native Library →   ~~Sop~~
                          System.loadLibrary("native Library")
                        }
② Declare a native method → public native void m1();
                     }

                     class Child
                     {
                       p.s.v.m(——)
                       {
③ Invoke a native method   Native n = new Native();
                           n.m1();
                     } }
```

→ for native methods implementation is already available in other languages and we are not responsible to provide implementation. Hence native method declaration should Compulsary Ends with ";"

ex:- ①
```
class Test
{
    public native void m1()
    {
    }
}
```
X

C.E:- native methods Can't have a body.

②
```
public native void m1();
```
✓

① → for native methods implementation should be available in Some other Languages where as for abstract methods implementation should not be available hence abstract-native Combination is illegal Combination for methods.

② → native methods Cannot be declared with Strictfp modifier because There no guarante that old language follows IEEE 754 Standard.

③ → Hence ~~abstract~~ native-Strictfp Combination is illegal for methods.

→ The main disadvantage of native keyword is ~~it~~ it breaks platform independent nature of Java. because we are depending on result of platform dependent languages.

⊙ "Synchronized" modifier :-

→ Synchronized is the modifier applicable for methods & Blocks. we can't declare class & variable with this keyword.

→ If a method (or) Block declared as Synchronized then at a time only one Thread is allowed to operate on the given Object.

→ The main advantage of Synchronized keyword is we can resolve data inconsistancy problems. But the main dis-Advantage of Synchronized keyword is it increases waiting time of thread and effects performance of the System. Hence, If there is no specific requirement it is never recommended to use synchronized keyword.

⊙ "transient" modifier :-

→ transient is the modifier applicable only for variables & we can't apply for methods & classes.

→ At the time of Serialization, if we don't want to save the value of a particular variable to meet security constraints, then we should go for transient keyword.

→ At the time of Serialization JVM ignores the original value of transient variable & default value will be Serialization.

⊙ "Volatile" modifier :-

→ Volatile is the modifier applicable only for variables but not for methods & classes.

→ If the value of a variable keep on changing such type of variables we have to declare with volatile modifier.

↳ If a variable declared as volatile then for every Thread a Seperate local Copy will be Created.

↳ Every intermediate modification Performed by that thread will takes place in local Copy instead of master Copy.

↳ Once the value got finalized just before terminating the thread the master Copy value will be updated with local stable value.

↳ The main advantage of volatile keyword is we can ~~achieve~~ resolve data inconsistency problems.

↳ But the main disadvantage of volatile keyword is, Creating & maintaining a Seperate Copy for every thread, increases Complexity of the programing & effects performance of the System. Hence, if there is no Specific requirement it is never recommended to use volatile keyword, & it is almost outdated keyword

↳ volatile variable means it's value keep on changes where as 'final' variable means its value never changes. Hence final-volatile Combination is illegal Combination for variables.

Conclusion:-

→ The only appliable modifier for local variables is 'final'

→ The modifiers which are appliable only for variables, but not for classes & methods are. Volatile & transient

→ The modifiers which are applicable only for methods but not for classes & variables native & synchronized.

→ The modifiers which are applicable for top Level classes, methods & variables are public, <default>, final

Modifier	Classes		methods	variables	blocks	interfaces	enum	Constructors
	Outer	Inner						
public	✓	✓	✓	✓	×	✓	✓	✓
<default>	✓	✓	✓	✓	×	✓	✓	✓
Private	×	✓	✓	✓	×	×	×	✓
Protected	×	✓	✓	✓	×	×	×	✓
final	✓	✓	✓	✓	×	×	×	×
abstract	✓	✓	✓	×	×	✓	×	×
Static	×	✓	✓	✓	✓	×	×	×
Synchronized	×	×	✓	×	✓	×	×	×
native	×	×	✓	×	×	×	×	×
Strictfp	✓	✓	✓	×	×	✓	✓	×
transient	×	×	×	✓	×	×	×	×
Volatile	×	×	×	✓	×	×	×	×

→ The modifiers which are applicable for Inner classes but not for Outer Classes are private, protected, Static

Interfaces

(1) Introduction

(2) Interface declaration & Implementation
(a) extends Vs implements.

(3) Interface methods

(4) Interface Variables

(5) Interface Naming Conflicts

(1) method naming Conflicts

(2) Variable " "

(6) marker Interface

(7) Adapter class

(8) Abstract Class Vs Concrete Class vs interface.

(9) diff. b/w abstract class & interface

Interface:-

(*)→ Any Service requirement Specification (SRS) is Considerad as Interface.

→ from the client point of view ~~Can~~ an Interface defines the Set of Services what is Expecking.

→ from the Service provider point of view an Interface defines the Set of Services what is offering.

(**)→ Hence an Interface Considered as Contract b/w Client & Service provider.

Ex:-

→ By using Bank ATM GUI Screen, Bank people will hightLate the Set of Services what they are offering At the Same time The Same Screen describes the Set of Services what End-User is Expected. Hence this GUI Screen acts as Contract b/w ~~the~~ bank people & customers.

→ with in the Interface we Can't write any implementation because it has to highlight Just the Set of Services what we are offering or what you are Expecting. Hence every method present inside interface Should be abstract. Due to this interface is Considered as 100% pure abstract class

What is an Interface:-

→ Any Service requirement Specification (SRS) (or) Any Contract b/w Client & Service provider (or) 100% pure abstract class is nothing but an Interface.

→ The main Advantages of Interfaces are.

(i) we Can acheive Security because we are not highlighting our internal implementation.

(ii) Enhancement will become Very easy, because with out effecting outside person we Can change our internal implementation.

(iii) Two different Systems Can Communicate via Interface (A Java application Can talk with Mainframe System through Interface).

Declaration & Implementation of an Interface :-

→ We Can declare an Interface by using Interface keyword, we Can implement an Interface by using implements keyword.

Ex:-

```
interface Interf
{
  void m1();   // by default public abstract void m1();
  void m2();
}
abstract class ServiceProvider implements Interf
{
  → public void m1()
    {
    }
}
```

→ If a class implements an interface Compulsary we should provide implementation for every method of that interface otherwise we have to declare Class as abstract. Violation leads to Compile-time Error.

→ when ever we are implementing an interface method Compulsary it should be declared as public otherwise we will get CompiletimeError.

Extends Vs implements :-

1. A class Can extend only one class at a time.
2. A Class Can implement any no. of interfaces at a time.
3. A Class Can extend a Class and Can implement any no. of interfaces Simultaneously.
4. An Interface Can extend any no. of interfaces at a time.

ex:-

```
interface A
{
}
interface B
{
}
interface C extends A, B
{
}
```

Q) which of the following is True?

(1) A Class Can extend any no. of Classes at a time. X

(2) A Class Can implement only one Interface at a time. X.

(3) A class Can extend a Class ~~and~~ or Can implement an interface but not both Simultaneously X

(4) An Interface Can extend only one Interface at a time X

(5) An Interface Can implement any no. of classes at a time X

(6) None of the above ✓

Q) Consider the expression

$\boxed{\text{X extends Y}}$ — for which of the following possibilities this Expression is True?

① Both should be Classes

② Both should be interfaces

✓③ Both Can be either Classes or interfaces

④ No Restriction.

A:

① X extends Y, Z

(a) X, Y, Z should be interfaces

② X extends Y implements Z

X, Y → Classes

Z → interfaces

③ X implements Y extends Z ✗

→ C.E

Interface methods:-

whether we are declaring or not, every interface method is by-default, public & abstract

eg:-
```
interface Interf
{
    void m1();
}
```

public:-

→ To make this method availability for every implementation class.

abstract:-

Because interface methods Specifies requirements but not implementation.

Hence the following method declarations are equal inside interface.

(1) void m1(); ✓

(2) public void m1(); ✓

(3) abstract void m1(); ✓

(4) Public abstract void m1(); ✓

→ As every interface method is bydefault public & abstract the following modifier are not applicable for interface methods.

(1) private
(2) protected
(3) <default>
(4) final
} ✗

(5) Static
(6) Strictfp
(7) Synchronized
(8) native
} ✗

→ which of the following method declaration are valied inside interface:

(1) public void m1() {} ✗

(2) public static void m1(); ✗

(3) public synchronized void m1(); ✗

(4) private abstract void m1(); ✗

(5) public abstract void m1(); ✓

interface variables:-

→ An interface can contain variables the main purpose of these variables is to specify.

Constents at requirement Level:-

→ Every interface variable is always public, static, final whether we are declaring (or) not.

```
interface Inter
{
 int x=10;
}
```

public:- To make this variable available for every implementation class.

static:- without existing object also implementation class can access this variable.

final:- implementation class can access this variable but can't modify.

→ Hence inside interface the following declaration are valid & equal.

1) int x=10;

2) public int x=10;

3) public static int x=10;

4) public static final int x=10;

5) public static int x=10;

6) final int x=10;

7) public final int x=10;

8) Static final int x=10;

→ As interface variables are public static & final we can't declare with the following modifiers.

(1) private

(2) protected

(3) <default>

(4) transient

(5) volatile.

→ for the interface variable compulsary are should perform initialization at the time of declaration only otherwise will get compile time Error.

(*)
```
interface Interf
{
    int x; X        C.E:- = Expected.
}
```

→ which of the following variable declarations are allowed inside interface.

(1) int x=10; ✓

(2) int x; X

(3) private int x=10; X

(4) public int x=10; ✓

(5) transient int x=10; X

(6) volatile int x=10; X

(7) public static final int x=10; ✓

→ Inside implementation Classes we Can access interface variables but we Can't modify These values.

Ex:-

```
interface Interf
{
    int x =10;
}
```

```
Class Test implements Interf
{
 p.s.v.m(String[] args)
 {
     x=888;  ✗
     S.o.pln(x);
 }
}
          C.E.
```

```
Class Test implements Interf
{
 p.S.v.m (String[] args)
 {
     int x =88;
     S.o.pln(x); 88
 }
}
        ✓
```

Interface nameing Conflicts :-

① method naming Conflicts :-

Case1:-

→ If Two interfaces Contains a method with Same Signature & Same return type in the implementation class we Can provide implementation for only one method.

Ex:-

```
interface Left
{
  Public void m1();
}
```

```
interface Right
{
  public void m1();
}
```

```
Class Test implements Left, Right
{
    public void m1()
    {
    }
}
```

✓

Case2:-

→ If Two interfaces Contains a method with Same name but different args then, in the implementation class we have to provide implementation for both methods & these methods are Considered as overloaded methods.

Ex:-

```
interface Left                  | interface Right
{                               | {
    public void m1();           |   public void m1(int i);
}                               | }
```

```
            Class Test implements Left, Right
            {
                public void m1()
              | {
              | }
Overloaded    | public void m1(int i)
methods       | {
              | }
            }
```

Case 3 :-

→ If Two interfaces Contains a method with Same Signature but different returntypes. Then it is impossible to implement both interfaces at a time.

Ex :-

```
interface Left              |   interface Right
{                           |   {
  public void m1();         |     public int m1();
}                           |   }
```

→ We Cant write any Java Class which implements both interfaces Simultaneously.

Q) is It possible A Java class Can implement any no. of interfaces Simultaneously.

A) yes, Except if Two interfaces Contains a method with Same Signature but different return types.

(2) Variable naming Conflicts :-

```
interface Left
{
   int x=888;
}
interface Right
{
  int x=999;
}
```

```
Class Test implements Left, Right
{
  p.s.v.m(——)
  {
   S.o.pln(x);
  }
}
```

C.E:- reference to x is abiguoues.

→ There may be a chance of 2 interfaces contains avaiable with same name & may rise variable naming conflicts But we can resolve these naming conflicts by using Interface names.

```
S.o.p(Left.x) ; 888
S.o.p(Right.x) ; 999
```

** Marker Interface :-

Ex :- Kenya

→ If an interface wont contain any method & by implementing that interface if own objects will get ability such type of interfaces are called marker interface (or) Tag interface (or) ability interface.

Ex:- Serializable, Clonable, RandomAccess, Single Thread Model.

→ these interfaces are marked from some ability.

Ex:- By implementing Serializable interface we can send object across the N/w and we can save state of object to a file. This extra ability is provided through Serializable interface.

Ex:- By implementing Cloneable interface our Object will be in a position to provide exactly duplicate Objects

Q) Marker interface wont contain any method then how the Objects will get that Special ability?

A) JVM is responsible to provide required ability in marker interfaces.

Q) Why JVM is providing required ability in marker interface?

A) To reduce Complexity of The programing.

Q) Is it possible to Create our own Marker Interface?

A) Yes, But Customization of JVM is required.

Ex:- Sleepable, Eatable, Jumpable, Lovable, Funnable.

Adapter class:-

→ Adapter class is a Simple java class that implements an interface, an interface only with Empty implementation.

```
interface X
{
    m1();
    m2();
    |
    |
    m1000();
}
```

```
abstract class AdapterX implements X
{
    m1() {}
    m2() {}
    |
    |
    m1000() {}
}
```

If we create an object for this Empty result So for this class are declare as abstract. by default abstract

→ If we implement an interface directly ~~coo~~ Compulsary we should provide implementation for every method of that interface, whether we are intrested or not & whether it is required (or) not. It increases length of the Code, So that readability will be reduced.

```
Class Test implements X
{
  m1() {}
  m2() {}
  m3()
  {
    ==
  }
  :
  m100() {}
}
```

If we extends adapter class instead of implementation interface directly then we have to provide implementation of only for required method but not all this approach reduce Length of the Code & improves readability.

```
⇒ Class Test extends Adapter X
{
  m4()
  {
    ==
  }
}
```

Concreate class Vs abstract Class Vs interface :-

→ we don't know any thing about implementation Just we have requirements Specification, then we should go for interface

Ex1. Servlet.

→ we are talking about implementation but not Completly (Just partially implementation) Then we should go for abstract Class.

Ex:- Generic - Servlet
HTTP_Servlet

→ we are talking about implementation Completly & ready to provide Service, Then we should go for Concrete Class.

Ex:- Our own Servlet.

Difference b/w interfaces & abstract class:-

interface	abstract class
1) If we don't know any thing about implementation Just we have requiremement Specification. Then we should go for interface.	1) If we are talking about implementation but not Completly (partially implementation) then we should go for abstract class.
2) Every method present inside interface is by default public & abstract.	2) every method present inside abstract class need not be public & abstract. we can take concrete methods also.
3) The following modifiers are not allowed for interface methods: strictfp, protected, static, native private, final, synchronized,	3) There are no restrictions for abstract class method modifier i.e, we can use any modifier.

4) every variable present inside interface is public, static final, bydefault wheather we are declare (or) not	4) abstract class variables need not be public, final static.
5) for the interface variables we can't declare the following modifiers private, protected, transient, volatile	5) there are no restriction for abstract class variable modifiers.
6) for the interface variables compulsary we should perform initialization at the time of declaration only	6) for the abstract class variables there is no restriction like performing initialization at the time of declaration
7) Inside interface we can't take instance & static blocks.	7) Inside abstract class we can take static block & instance blocks.
8) Inside Interface we can't take constructor.	8) Inside abstract class we can take constructor.

Q) Inside abstract class we Can take Constructor but we Can't Create an object of abstract class, what is the need?

A) → abstract Class Constructor will be executed whenever we are Create child class object to perform initialization of parent class instance variable at parent Level only and this Constructor meant for child object Creation only

Q) Inside Interface every method should be abstract where as in abstract class also we Can take only abstract methods Then what is the need of interface?

A) → Interface purpose we Can replace abstract class but it is not a good programing practice we are miss using the role of abstract class.

→ we should bring abstract class into the picture whenever we are talking about implementation.

ohm Sai Ram prabu.

28/4/11

Oops Concept

×o×

① Data Hiding :-

→ Hiding of the data, So that outside person Cant access our data directly.

→ By using private modifier we Can implement Data Hiding.

Ex:-

```
Class Account
{
    private double balance = 1000;
}
```

→ The main Advantage of Data Hiding is we Can acheive Security.

2) Abstraction :-

→ Hiding internal implementation details & just highylate the set of Services what we are offering, is called "Abstraction."

ex:-

↳ By Bank ATM machine, Bank people will highLate the Set of Services what they are offering without Highlating internal implementation. This Concept is nothing but Abstraction.

→ By Using interfaces & abstract classes we Can acheive abstraction.

→ The main Advantages of Abstraction are.

1) We Can acheive Security as no one is allowed to Know our internal implementation.

2) With out effecting outside person we Can change our internal implementation Hence Enhancement will become Very easy.

→ The main disadvantage of Encapsulation is it increases the Length of the Code & slows down Execution.

4) Tightly Encapsulated class :-

→ A class is said to be tightly encapsulated iff every data member declared as the private.

→ wheather the class contains getter & setter methods are not & wheather those methods declared as public or not these are not required to check.

ex:-

```
Class A
{
  Private int balance;
  Public int getBalance()
  {
    return balance();
  }
}
```

ex:- which of the following classes are Tightly Encapsulated.

```
✓  class A
   {
    private int x=10;
   }
x. Class B extends A
   {
    int y=20;
   }
✓  Class C extends A
   {
    Private int z=30;
   }
```

③ It improves modularity of the application. meaning?

3) Encapsulation :-

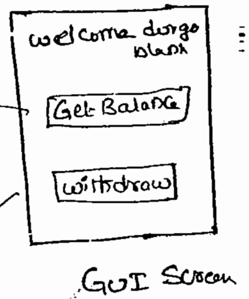
↳ Encapsulating data & corresponding methods (behaviour) into a Single module is Called "Encapsulation".

→ If any Java class fallows Data Hiding & Abstraction such type of class is Said to Encapsulated class.

Encapsulation = DataHiding + Abstraction

Ex:-

```
        class Account
        {
          private double balance;
          public double getBalance()
          {
             // validate user
             return balance;
          }
          public void setBalance(double balance)
          {
             //validate user
             this.balance = balance;
          }
        }
```



→ Hiding data behind methods is the Central Concept of Encapsulation

→ The main advantages of Encapsulation are ① we Can acheive Security.

② Enchancement will become Very easy.

③ improves modularity of the application.

Ex 3:- Which of the following classes are Tightly Encapsulated.

Ex:

Q) by default what modifier to variables?

```
     Class A
  ×  {
       int x=10;
     }
     Class B extends A
  ×  {
       private int y=20;
     }
     Class C extends A
  ×  {
       private int z=30;
     }
```

Conclusion:-

→ If parent class is not tightly Encapsulated then no child Class is Tightly Encapsulated.

5) IS-A Relationship :-

→ It is also known as Inheritance

→ By using extends Keyword we Can implement IS-A Relationship

→ The main advantage of IS-A Relationship is Reusability of the Code.

Ex:-

```
Class P
{
  public void m1()
  {
    ===
  }
}
Class C extends P
{
  public void m2()
  {=
  }
```

```
Class Test
{
 P.S.V.m(String[] args)
 {
```

Case1: P p = new P();

p.m1(); ✓

p.m2(); X → C.E!- Cannot find Symbol

Symbol : method m2()

location : class P

Case 2! C c = new C();

c.m1(); ✓

c.m2(); ✓

* **Case3!.** P p1 = new C();

p1.m1(); ✓

p1.m2(); X C.E!

* **Case 4!.** C c1 = new P(); X C.E!- incompatible types

found : P

required : C

Conclusion(s)!-

① what ever the parent class has by default available to the child. Hence ~~and~~ ~~with the~~on child class reference we can call both parent & child class methods.

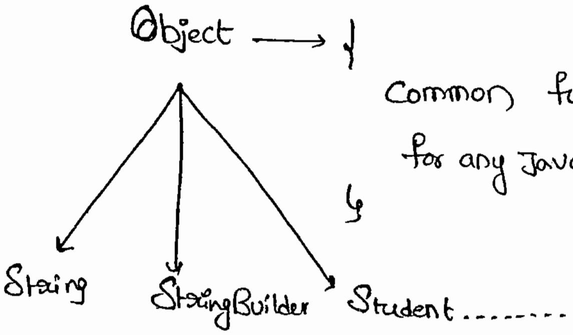
② what ever the child has bydefault not available to the parent hence on the parent class reference we can call only parent class methods & we cant call child specific methods.

③ parent class reference can be used to hold child class Objects by using that reference we can call only parent class methods but we can't call child specific methods.

④ We can't use child class reference to hold parent class Objects.

Ex:-

① The Common functionality which is required for any java classes is defined in Object class and by keeping that class as Super class it's functionality by default available to every Java classes.



Ex:- The Common functionality which is required for all Exceptions & Errors is defined in Throwable class as Throwable is parent for all Exceptions & Errors, it's functionality will be available automatically to every child not required to rewrite.

Q) Do 'Throwable' has 'object' as parent class?
→ yes

→ Java won't provide Support for multiple inheritance but Through interfaces it is possible.

Ex!-

Class A extends B , C
{
}
✗
C.E!.

But

interface A extends B C
{
}
✓

→ Every Class in Java is the child Class of Object.

→ If our class doesn't extend any other class then only it is the direct child Class of Object.

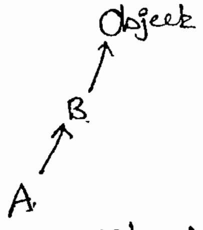
ex:-

```
Class A
{
  =
}
```

→ If our Class extend any other Class then our Class is not directly child class of Object.

ex:-

```
class A extends B
{
   =
}
```

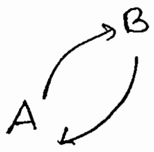


(multi Level inheritance).

→ Cyclic inheritance is not allowed in Java

ex:- ①

```
Class A extends B
{
}
Class B extends A      ✗
{
}
```



C.E:- Cyclic inheritance involving A

②

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

6) Has - A Relationship :-

→ Has- A Relationship is also Known as "Composition or Aggregation".

→ There is no Specific Keyword to implement Has - A Relationship the mostly we are using 'new keyword'.

→ The main advantage of Has- A Relationship is Reusability or (Code Reusability)

ex:-

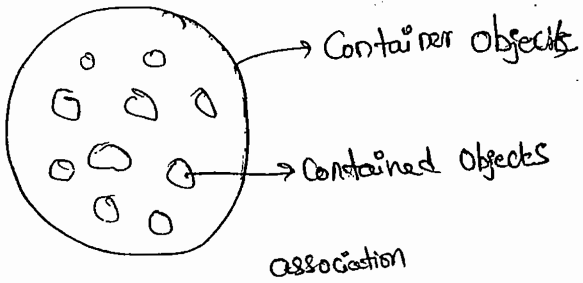
```
Class Car                            Class Engine
{                                    {
  Engine e = new Engine();             // Engine Specific functionality
}                                    }
```

Class Car has Engine reference.

→ The main disadvantage of Has-A Relationship is it increases dependency b/w the classes and creates maintaince problems.

Composition Vs Aggregation :-

→ In the Case of Composition whenever Container objects is destroyed all Contained objects will be destroyed automatically. i.e, without Existing Container object there is no chance of existing contained object i.e Container & Contained objects having Strong association



ex:-

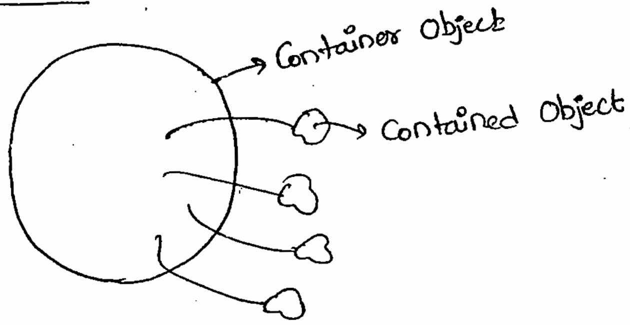
→ University is Composed of Several departments

→ whenever you are closing University automatically all departments will be closed. the relationship b/w University object & department object is strong association which is nothing but Composition.

Aggregation :-

→ whenever Container object destroyed, there is no garanenty of destruction of Contained objects i.e, without existing Container object there may be a chance of existing Contained object. i.e, Container object just maintains References of Contained objects.

This relationship is Called weak association which is nothing but "Aggregation".



ex:-

→ Several professors will work in the department

→ whenever we are closing the department still there may be a chance of existing professors. the relationship b/w department & professor is Called weak association which is nothing but Aggregation.

```
        public void m1(int i)
        {
          S.o.pln("int-arg");
        }
        public void m1(float f)
        {
          S.o.pln("float-arg");
        }
      p.s.v.m (______)
      {
       Test t = new Test();
          t.m1();    // no-arg
          t.m1(10);  // int-arg
          t.m1(10.5f); // float-arg
      }
    }
```

*→ In overloading method resolution always takes care by compiler based on reference type. Hence overloading is also considered as Compiletime polymorphism (or) Static polymorphism (or) Early Binding

→ In overloading reference type will play very important role & runtime object will be dummy.

Case1:-

*Automatic promotion in overloading:

→ In overloading method resolution, if the matched method with specified argument type is not available then compiler won't raise any Error immediately. first it promotes that arrguement to the next Level and checkes for matched method.

→ If the matched method is available then it will be Considered and if it is not available then Compiler once again promotes this arrguement to the next Level.

→ This process will be Continued untill all possible promotions after Completing all promotions Still if the matched method is not available then only we will get C.E.

→ this ~~fall~~ is Called Automatic promotion in overloading.

→ The following are Various possible promotions in Overloading.

$$\begin{tikzcd}
byte \arrow[r] & short \arrow[dr] & & & \\
& & int \arrow[r] & long \arrow[r] & float \arrow[r] & double \\
& char \arrow[ur] & & &
\end{tikzcd}$$

Case1:-

Ex:-

```
Class Test
{
  public void m1(int i)
  {
     S.o.pln("int-arg");
  }
  public void m1(float f)
  {
     S.o.pln("float-arg");
  }
  P.S.V.m(String[] args)
  {
     Test t = new Test();
```

Method Signatuae :-

→ method Signature Consists of name of the method & arrguement List.

Ex:- public void m1 (int i, float f)

⇓

m1(int , float)

→ In Java return type is not part of method Signature.

→ Compile will always use method Signature while reasolving method Calls

→ With in the Same Class Two methods with the Same Signature not allowed Otherwise we will get Compiletime Error.

Ex:-
```
Class Test
{
  public void m1(int i)
  {
  }
  public int m1(int i)
  {
    return 10;
  }
}
```

m1(int) is the method Signature.

```
Test t = new Test()
t1.m1(10);
```

C.E :- m1(int) has already defined in Test

Overloading

Overloading:-

→ Two methods are Said to overloaded iff method names are Same but arguements are different.

→ Lack of overloading in 'c' increases Complexity of the program. In C, language if there is a change in method arguement type Compulsary we should go for new method name.

ex:-

abs()	→ int
labs()	→ long
fabs()	→ float
⋮	

→ But in Java Two methods having the Same name with different arguements is allowed & these methods are Considered as overloaded methods. ex:-

abs(int)
abs(long)
abs(float)
⋮

→ Having overloading Concept in Java Simplifies the programming

Ex:-

```
Class Test
{
  Public void m1()
  {
    S.o.pln("no-arg");
  }
```

Case:-
→ In Overloading most specific version will get highest priority.
what does it mean?

Case 2:-

Ex!-

```
Class Test
{
  Public Void m1(StringBuffer Sb)
   {
      S.o.pln("StringBuffer-arrg");
   }
  Public void m1(String s)
   {
      S.o.pln("string-version");
   }
}
  public . s . v.m (——)
{
  Test t = new Test();
  t.m1(new SB("durga")); //StringBuffer-args
  t.m1("durga"); // String version
  t.m1(null); // C.E!- reference m1() is ambiguity.
 }
}
```

By default 'String' constant of String class object type like 'integral constant of 'int' "double floating literal"

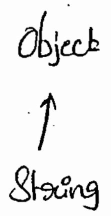
```
        t.m1('a'); // int-arg
        t.m1(10l); // float-arg
        t.m1(10.5); x C.E:-
                          Cannot find Symbol
        }                 Symbol : method m1(double)
    }                     location : Class Test
```

*Case 2:-

→ In overloading method resolution child-argument will get more priority than parent arrguement.

Ex:-

```
    Class Test
    {
①    public void m1(Object o)
      {
        S.o.pln("Object version");
      }
②    public void m1(String s)
      {
        S.o.pln(" String Version");
      }
     p.s.v.m (——)
     {
       Test t = new Test();
       t.m1(new Object()); // object-version
       t.m1("durga"); // String-version (suppose ② statement takes // ~~String~~ The o/p is Object
       t.m1(null); ? // String-version
    } }
```



→ ~~Here~~ Hence overriding is also known as "runtime polymorphism (or) dynamic polymorphism (or) Late binding".

→ Overriding method resolution is also known as "Dynamic method dispatch".

Rules for Overriding :-

① In overriding method names & arguements must be matched i.e, method Signatures must be matched.

②. In overriding return type must be matched, But this rule is applicable untill 1.4 version, from 1.5 Version onwards Co-variant (ఒక దాని బదులు మరొకటి) return types are allowed. according to this, child method return type need not be same as parent method return-type. its child classes also allowed.

Ex:

```
Class P
{
  Public Object m1()
  {
   return null;
  }
}
Class C extends P
{
  Public String m1()
  {
   return null;
  }
}
```

✓ It is valid in 1.5v, But invalid in 1.4V

Ex :-

Parent method return type	Object ✓	Number ✓	String ✓	double
	↓	↓	↓	↓
Child method return types	String ✓	Integer ✓	Object ✗	int ✗

→ Co-varient return type Concept is applicable only for object type but not for primitive types.

③ we Can't override parent class final method. But we can use it as it is

④ private methods are not visible in child classes Hence overridding Concept is not applicable for private methods.

⑤ → Based on our requirement we Can declare the Same Parent class private method in Child Class also it is valid but it is not overriding.

ex:-

```
Class P
{
 private void m1()
 {
   =
 }
}
Class C extends P
{
 private void m1()   ✓
 {
   =
 }
}
```

not overriding

(*) Compiles fine but not overriding.

→ for parent class abstract methods we should override in Child class to provide implementation.

(*) → we Can override parent class non-abstract method as abstract in child class To stop parent class method implementation availability to the child classes.

ex:-

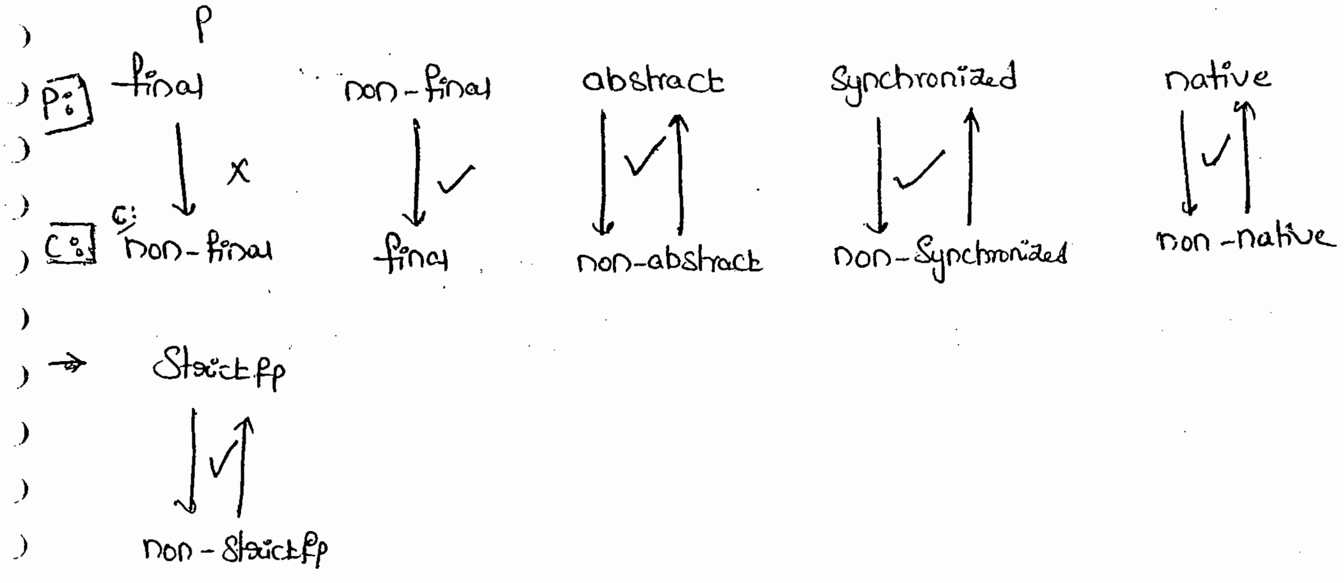
```
Class P
{
 Public void P()
 {
 }
}
abstract class C extends P
{
 Public abstract void P(); ✓
 {
 }
}
```

→ The following modifiers won't play any restrictions in overriding

① native.

② Synchronized

③ strictfp



→ while overriding we can't decrease scope of the modifier but we can increase the following are various acceptable overridings

Private < default < protected < public

P)	Public	protected	default	private
	↓	↓	↓	↓
C)	public.	Protected/public	default/protected/public	Private method Can't be override

Eg:-

```
Class P
{
  public void m1() { }
}
Class C extends P
{
  protected void m1()  X
  {
  }
}
```

C.E:-

m_1 in C Can't override in C

→ This rule is applicable while implementing interface methods also.

→ whenever we are implementing any interface method Compulsary it should be declared as public. because Every interface method is public by default.

Ex:-

```
interface Interf
{
   void m1();
}
Class Test implements Interf
{
   void m1()        X C.E:-
   {
   }
}
```

if we declare public we won't get any C.E ←

⊛ → If child class method throws some checked Exception then Compulsary Parent class method should throw the same checked Exception or it's Parent class Exception. otherwise we will get C.E.

→ But there is no rule for unchecked Exception.

Ex:-①

```
Class P
{
  Public void m1()
  {
  }
}
Class C extends P
{
  Public void m1() throws Exception    ✗
  {
  }          C.E:-  m1() in C can't override m1() in P;
}                   Overridden method doesnot throw Exception.
```

Ex②:-

ⓐ ✓
P : public void m1() throws IOException
C : public void m1()

ⓑ ✗
P : public void m1()
C : public void m1() throws IOException

ⓒ ✓
P : public void m1() throws Exception
C : public void m1() throws IOException

ⓓ ✗
P : public void m1() throws IOException
C : Public void m1() throws Exception

⑤ P: public void m1() throws IOException

✓ C: public void m1() throws FileNotFoundException, EOFException

⑥ P: public void m1() throws IOException

✗ C: public void m1() throws EOFException, InterruptedException

⑦ P: public void m1() throws IOException

✓ C: public void m1() throws AE, NPE

⑧ P: public void m1()

✓ C: public void m1() throws AE, NPE

Overriding w.r.t static method :-

→ We Can't override a static method as non-static.

Ex:-

```
Class P
{
    Public Static Void m1()
    {
    }
}
Class C extends P
{
    public void m1()
    {
    }
}
```

✗

static
↓✗↑
non-static

C.E:- m1() in C Can't override m1() in P;
overriden method is static.

$10^{8}{}_{14}$

→ Similarly, we Can't override non-Static method as Static

→ If both parent & child Class method ~~Class~~ are Static Then we wont to get any C.E it Seems to be overriding is happen, but it is not overriding. it is "Method Hiding".

Ex:-

```
Class P
{
    Public Static void m1()
    {
    }
}
Class C extends P
{
    Public Static void m1()
    {
    }
}
```

it is not overriding it is "method Hiding"

Method Hiding :-

→ All rules of Method Hiding are Exactly Same as Overriding Except the following difference.

method hiding	Overriding
1) Both methods Should be Static	1) Both methods should be non-Static.
2) method resolution takes Care by Compiler based on Reference type.	2) metho resolution always takes Care by JVM based on Runtime object.
3) It is Considered as Compiletime Polymorphism or Static polymorphism or early Binding	3) It is Considered as Runtime Polymorphism or dynamic polymorphism or Late Binding.

Ex:

method hiding

```
Class P
{
 Public Static void m1()
  {
   S.o.pln(" parent");
  }
}
Class C extends P
{
  Public Static void m1()
   {
    So.pln(" child");
   }
}

Class Test
{
 p.s.v.m (——)
  {
   P   p = new  P();
      p.m1();       → parent
   C   c = new  C();
       c.m1();   → child
    P  P1 = new  C();
       P1 . m1();     Parent
  }
}
```

→ If both methods are non-Static then it will become overriding in this Case the o/p is: Parent
Child
Child

Overriding w.r.t Var-arg methods :-

→ We Can't override a Var-arg method with general method. If we are trying to override it will become overloading but not overriding.

→ A Var-arg method should be overriden with Var-arg method only.

Ex:-

```
Class P
{
  public void m1(int... i)
  {
    S.o.pln(" parent");
  }
}
Class C extends P
{
  Public void m1(int i)
  {
    S.o.pln(" child");
  }
}
Class Test
{
  p.s.v.m(——)
  {
    P p =new P();
     p.m1(10); // Parent
    C c = new C();
     c.m1(10); //child
                         P p1= new C();
                           p1.m1(10); //parent
  }
}
```

overloading but not overriding

→ If both parent & child class methods are Var-arg then it will becomes overriding in this case o/p is parent child child ~~parent~~

Overriding w.r.t Variables:-

→ Overriding concept is not applicable for variables.

→ Variable resolution always takes care by compiler based on reference type. runtime object won't to play any role in variable resolution.

Ex:-

~~both static~~

```
Class P
{
  int x=888;
}
Class C extends P
{
   int x=999;
}
 Class Test
 {
   p.s.v.m(______)
   {
      P p=new P();
     S.o.pln(p.x); // 888
      C c=new C();
       S.o.pln(c.x); //999 ✓
      P p1=new C();
        S.o.pln(p1.x); 888.
   }
 }
```

both static o/p 888 | both instance o/p 888 | one static & one instance o/p 888

→ wheather The variables are Static or non-Static There is no Change in result.

difference b/w Overloading & Overriding :-

Property	Overloading	Overriding
① method names	must be Same	must be Same
② arguements	must be different (at least order)	must be Same (including order) .
③ method Signature	must be different	must be Same .
④ return type	No restructions	must be Same until 1.4V but from 1.5V onwards Co-varient return types are allowed .
⑤ private, Static & final methods	Canbe overloaded	Can't be overridden
⑥ access modifiers	No restructions	Scope we Can't decrease The Scope .
⑦ Throws Clause	No restructions	Size & level of checked exceptions we Can't increase But we Can decrease . But No restructions for unchecked Exceptions .
⑧ method resolution	Always takes Care by Compiler based on reference type	Always takes Care by JVM based on runtime Object
⑨ Also known as	Compile-time polymorphism (or) Static polymorphism (or) Early binding	runtime polymorphism (or) dynamic polymorphism (or) Late binding.

Note:-

→ In overloading we have to check only method names (must be Same) & arguements (must be diff.) All remaining teams like (return type, throws clause, Accessmodifiers e.t.c) are not required to check.

→ But in overriding we have to check each & every thing.

Q) Consider the following method declaration in parent class which of the following methods allowed in child class?

P : public void m1(int i) throws IOException

Overriding ✓ ① public void m1(int i)

overloading ✓ ② public void m1( ) throws Exception

overloading ✓ ③ public static int m1(double d) throws IOException

C.E × ④ public int m1(int i)

C.E × ⑤ public synchronized void m1(int i) throws Exception

overloading ✓ ⑥ public static void m1(int... i) throws Exception

C.E × ⑦ public native abstract void m1().throws Exception.

Polymorphism

↳ poly → many

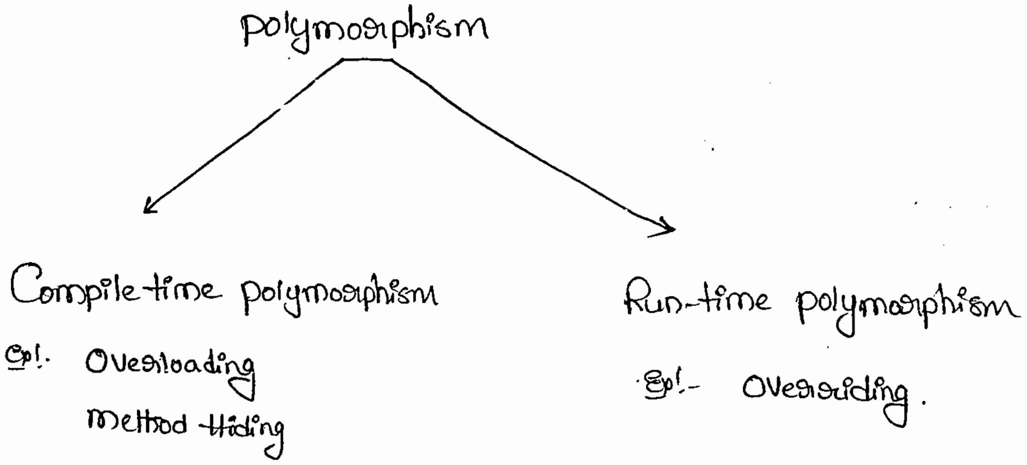
morphs $\xrightarrow{\text{means}}$ forms

i.e polymorphism means many forms

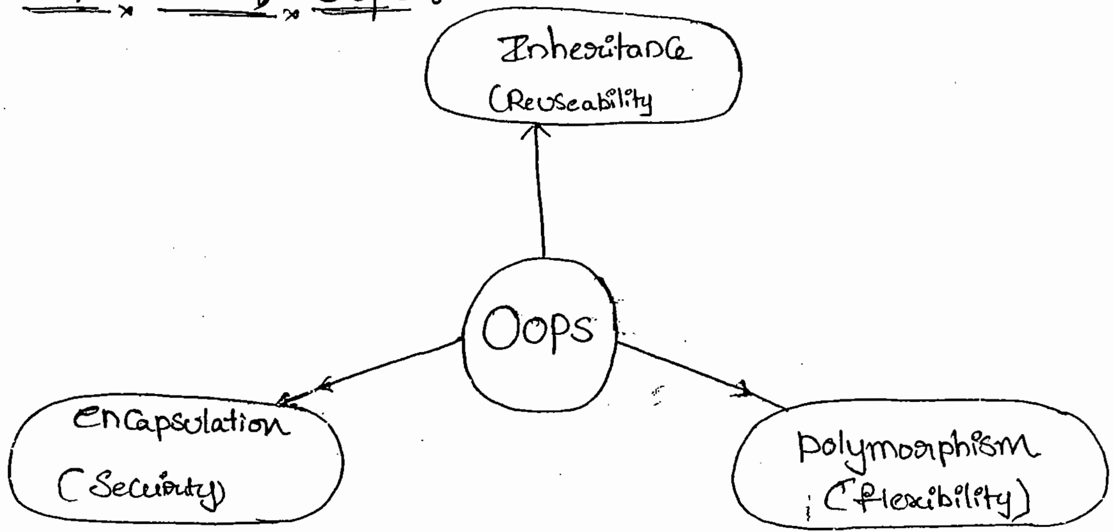
→ we can use same name to represent multiple forms in polymorphism.

Ex:-
→ In overriding we can have a method with one type of implementation in parent, but different type of implementation in child class.

→ There are 2 types of polymorphism.



3 Pillars of Oops :-



funny diffination of polymoriphism :-

→ A boy uses the word FRIENDSHIP to Starts LOVE, but girl uses the Same word to ~~Close~~ Ends. Same word but different attitudes. This behaviour is nothing but polymoriphism.

3) Static Control flow :-

Ex:-

```
        Class Base
        {
①        Static int x=10;          7
②        Static
          {
            m1();                   8
            S.o.pln("FSB");         10
          }
③        Public static void main(String[] args)
          {
            m1();                   13
            S.o.pln("main method"); 15
          }
④        Public static void m1()
          {
            S.o.pln(y);             9, 14
          }
⑤        Static
          {
            S.o.pln("SSB");         11
          }
⑥        Static int y = 20;        12
        }
```

x = 0[RIWO]

y = 0[RIWO]

x = 10 [R&W]

y = 20[R&W]

O/p:-

```
0
FSB
SSB
20
mainmethod.
```

Process:-

→ when ever we are trying to execute a Java class first that .class file should be loaded, at the time class loading the following actions will be performed automatically.

① identification of static members from Top to bottom. (1 to 6)

② Execution of Static variable assignments & Static blocks from top to bottom (7 to 12)

③ Execution of main method. (13 to 15)

Read Indirectly write only state (RIWOS)

→ If a variable is in Read indirectly write only state then we can't perform read operation directly otherwise we will get Compile-time Error saying "Illegal-forward-Reference".

Ex:-

```
Class Test
{
① Static int x=10;  ③
② Static
   {
     S.o.pln(x);
     System.exit(0);
   }
}
```

✓

O/P :- 10

```
Class Test
{
① Static
   {
     S.o.pln(x);  ③ ✗
   }
② Static int x=10;
}
```

C.E:- Illegal forward reference.

Static block :-

→ At the time of Class loading if we want to perform any activity we have to define that activity inside Static block because Static blocks will be Executed at the time of Class loading.

→ with in a class we Can take any no. of ~~Cla~~ Static blocks but all these Static blocks will be Executed from Top to bottom.

Ex(1) :-

→ After loading JDBC driver class we have to register driver with driverManager but every Driver class Contains a Static ~~meth~~ block to perform this activity at the time of Driver class loading automatically we are not responsible to perform register Explicitly.

```
Ex:   Class Driver
        {
          Static
           {
             Register ~~this~~ Driver with  DM
           }
        }
```

Ex(2) :- Advantage:

→ At the time of Class loading Compulsory we have to ~~t~~ load The Corresponding native libraries hence we can define this step inside Static block.

```
Ex1.    Class Native
         {
           Static
            {
              System.loadLibrary("native Library path");
            }
         }
```

Static Control flow in parents Child Classes:-

```
Class Base
{
①  Static int x=10;  ⑫
②  Static
   {
     m1();  ⑬
     S.o.pln(" Base SB");  ⑮
   }
③  Public Static void main(——)
   {
     m1();
     S.o.pln(" Base main");
   }
④  public static void m1()
   {
     S.o.pln(y);  ⑭
   }
⑤  Static int y= 20;  ⑯
}
Class Derived extends Base
{
⑥  Static int i=100;  ⑰
⑦  Static
   {
     m2();  ⑱
     S.o.pln(" DFSB");  ⑳
   }
⑧  Public static void main(——)
   {
     m2();  ㉓
     S.o.p(" Derived main");  ㉕
   }
(9)  Public Static Void m2()
     {
       S.o.pln(j);  (19)  (24)
     }
(10) Static
     {
       S.o.p("DSSB");  (21)
     }
(11) Static int j = 200;  (22)
}
```

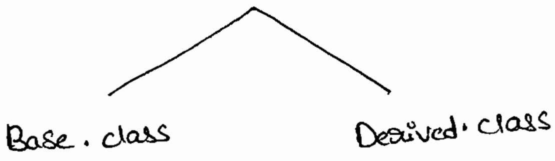
```
> java Derived
O/P:-   0
        Base SB
        0
        DFSB
        DSSB
        200
        Derived main
```

```
x = 0 [RIWO]
y = 0 [RIWO]
i = 0 [RIWO]
j = 0 [RIWO]
x = 10 [R W]
y = 20 [R & W]
i = 100 [R & W]
j = 200 [R & W]
```

```
> java Base
  0
  Base SB
  20
  Base main.
```

Process:-

> javac Derived.java



> java Derived

(1) Identification of Static member from parent to child [1 to 11]

(2) Execution of Static Variable assignments & static blocks from parent to child [12 to 22]

*(3) Execution of only child class main method [23 to 25]

(because main() method of parent class is overriding in child class, then, child-class main() method executed)

Process :-

→ whenever we are trying to load child class Then automatically parent class will be loaded to make parent class members available to the child class. Hence whenever we are Executing child class the following is the flow with respect to static members step.

(1) Identification of static members from parent to child

(2) Execution of static variable assignments & static blocks from parent to child.

(3) Execution of only child class main method. [If the child class won't contain main method Then automatically parent class main() method will be executed].

Note :-

When ever we are loading child class automatically parent class will be loaded. But when ever we are loading parent class child class wont be loaded.

Instance Control flow :-

```
   class Parent
   {
③   int x=10; ⑨
     {
④     m1();          ⑩
      S.o.p("FIIB"); ⑫
     }
⑤   Parent()
     {
      S.o.pln("Constructor"); ⑮
     }
①   Public static void main(String[] args)
     {
②     Parent p = new Parent();
      S.o.pln("main");
     }
⑥   Public void m1()
     {
      S.o.pln(y); ⑪
      }   ------------> instancblock
⑦   {
      S.o.pln("SIIB"); ⑬
     }
⑧   int y=20; ⑭
   }
```

x=0 [RIWO]
y=0 [RIWO]
x=10 [R W]
y=20 [R W]

O/P :-

```
0
FIIB
SIIB
Constructor
main
```

Process :-

→ whenever we are Creating an Object the following Sequence of events will be performed automaticaliy.

(1) Identification of instance member from Top to bottom [1 to 8]

(2) Execution of instance variable assignments & instance blocks from top to botom [9 -14].

(3) Execution of Constructor [15]

* Note :-

→ Static Control flow is only one time activity and it will be performed at the time of class loading But instance control flow is not one time activity for every object Creation it will be executed.

Instance Control flow from parent to child :-

```
Class Parent
{
③ int x=10; ⑮
④ {
    m1(); ⑯
    S.o.pln(" parent"); ⑱
  }
⑤ parent()
  {
   S.o.pln(" parent Constructor"); ⑳
  }
```

```
(1) public static void main(------)
    {
(20)  Parent p = new Parent();
      S.o.pln(" ~~child~~ Parent main");      (21)
    }
(6) public void m1()
    {
      S.o.pln(y);  (17)
    }
(7) int y = 20;  (19)
    }

    Class child extends Parent
    {
(10)  int i = 100;      (22)
      {
(11)    m2();      (23)
        S.o.pln(" CIIB");  (25)
      }
(12)  Child()
      {
        S.o.pln(" child Constructor");  (28)
      }
(8)   Public Static void main (------)
      {
(9)     child c = new child();
        S.o.pln(" child main");  (29)
      }
(13)  public void m2()
      {
        S.o.pln(i);  (24)
      }
```

```
0
parent
Parent Constructor
```

```
0
CIIB
CSIIB
Child Constructor
Child main.
```

```
(14) {
        S.o.pln(" CSIIB"); (26)
     }
     int j = 200; (27)
  }
```

Process :-

→ when ever ~~Sequence of~~ we are Creating child class object the following Sequence of ~~execute~~ events will be performed automatically.

(1) Identification of instance member from parent to child.

(2) Execution of instance variable assignments & instance blocks only in parent class.

(3) Execution of Parent class Constructor.

(4) Execution of instance variable assignments & instance blocks only in child class.

(5) Execution of child class Constructor.

```
>java child
> java parent
```

Constructors :-

→ Object Creation is not enough Compulsary we should perform initialization then only that Object is in a position to provide responce properlly.

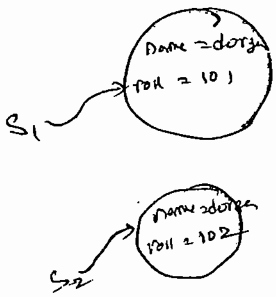
→ when ever we are Creating an Object Some peace of the Code will be executed automatically to perform initialization this peace of Code is nothing but Constructor. Hence the main objective of Constructor is to perform initialization for the newly Created Object.

ex:-

```
   Class Student
   {
①   int rollno;
②   String name;
    Student (String name , int rollno)
    {
        this.name = name;
        this.rollno = rollno;
    }
   Public static void main (String[] args)
   {
     Student s1 = new Student (" durga ", 101);
     Student s2 = new Student (" raghu ", 102);
   }
  }
```

0.

~~[illegible]~~ null.



Instance block Vs Constructor :-

→ At the time of object Creation if we want to perform initialization of instance variable then we should go for Constructor.

→ other than initialization activity if we want to perform any activity at the time of Object Creation then we Should go for instance block.

→ we Can't replace Constructors with instance block because Constructor Can take arguement where as instance block Can't take arguements.

→ Similarly we Can't replace instance block with Constructor because a class Can Contain more than one Constructor. if we want to replace instance block with Constructor then in every Constructor we have to write instance block Code because at runtime which Constructor will be Called we can't Expect. It results duplicate & Creates maintaince problems.

Ex!-

```
class Test
{
    static int Count =0;
    Test ()
    {
        Count ++;
    }
    Test (int i)
    {
        Count ++;
    }
    p.s.v.m (———)
    {
        Test t1 = new Test();
        Test t2 = new Test(10);
```

only once required

if we Create instance

Rules to define Constructors :-

1) the name of the class & name of the Constructor must be matched.

2) return type Concept is not applicable for Constructor even void also.

By mistake if we declare return type for the Constructor we wont get any Compiletime (or) runtime Errors, because Compiler treats it as method.

ex.

```
class Test
{
    void Test()  ⟶  It is a normal method but not Constructor
    {
    }
}
```

It is legal (for Stuppid to have a method whose name is exactly Same as class name).

(3) The only applicable modifiers for Constructors are "public, private, protected, <default> [PPPD]", if we are trying to use any other modifier we will get Compile-time Error Saying "modifier xxxx is not allowed here".

└→ static/final/strictfp ...

ex.

```
class Test
{
    final Test()
    {
    }        ✗
}               C.E: modifier final is not allowed here.
```

Singleton classes :-

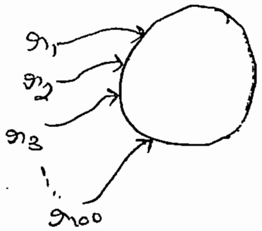
→ for any java class if we are allowed to create only one Object such type of class is called "Singleton class".

ex:- Runtime, ActionServlet (Structs 1.x)

BusinessDeligate (EJB), ServiceLocator (EJB) ---- e.t.c

→ The main advantage of Singleton is, instead of Creating a Separate Object for every requirement we can create a single object and reuse the same object for every requirement this approach improves memory utilization & performance of the system.

```
Runtime r1 = Runtime.getRuntime()
Runtime r2 = Runtime.getRuntime()
                 ↳ Class   ↳ Static method
   :
Runtime r100 = Runtime.getRuntime()
```

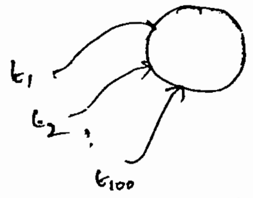


Creation of our own Singleton Class :-

→ we can create our own Singleton classes also for this we have to use private Constructor & factory method.

ex:-
```
class Test
{
    private static Test t;
    private Test()
    {
    }
    public static Test getInstance()
    {
```

```
    if (t == null)
    {
      t = new Test();
    }
    return t;
  }
  public Object clone()
  {
    return this;
  }
}
```



```
Test t1 = Test.getInstance();
Test t2 = Test.getInstance();
  :
Test t100 = Test.getInstance();
Test t101 = Test.clone();
```

factory method:-

→ By using class name if we call any method & return same class object. Then that method is consider as factory method.

Ex:-

```
Runtime r = Runtime.getRuntime();
```

→ factory method

```
DateFormat df = DateFormat.getInstance();
```

↳ factory method.

```
Test t = Test.getInstance();
```

↳ factory method

→ Similarly we can Create Doubleton, Threebleton ----- xxxton Classes.

How to Create Doubleton class :-

Ex:-

```
Class Test
{
  private Static Test t1;
  private Static Test t2;
  private Test()
  {
  }
  Public Static Test getInstance()
  {
   if (t1 == null)
   {.
     t1 =new Test();
     return t1;
   }
   else
   if (t2 == null)
   {
     t2 = new Test();
     return t2;
   }
  else
  {
    if (math.random() < 0.5)
     return t1;
    else
     return t2;
  }
}
```

Rule :-

Default Constructor :-

→ If we are not writing any Constructor then Compiler will always generate default Constructor.

→ If we are writing atleast one Constructor Then Compiler won't generate default Constructor.

→ Hence a class Can Contain either programmer writen Constructor (or) Compiler generated Constructor but not both Simultaniously.

Prototype of default Constructor :-

1) It is always no arguement Constructor.

2) The access modifier of default Constructor is Same as class modifier but this rule is applicable public & <default>.

3) It Contains only one line, It is a no arguement Call to Super class Constructor.

```
Test()
{
Super();
}
```

Programmers Code | Compiler generated Code

(1)
```
class Test
{
}
```

(1)
```
class Test
{
  Test()
  {
   Super();
  }
}
```

(2)
```
public class Test
{
}
```

(2)
```
public class Test
{
  public Test()
  {
    Super();
  }
}
```

(3)
```
Class Test
{
  void Test()
  {
  }
}
```

→ It is not a constructor
It is a normal method

(3)
```
Class Test
{
  Test()
  {
   Super();
  }
  void Test()
  {
  }
}
```

(4)
```
Class Test
{
  Test()
  {
  }
}
```

(4)
```
Class Test
{
  Test()
  { Super();
  }
}
```

(5)
```
Class Test
{
  Test()
  {
   this(10);
  }
  Test(int i)
  {
  }
}
```

(5)
```
Class Test
{
  Test()
  {
   this(10);
  }
  Test(int i)
  {
   Super();
  }
}
```

```
(6)   Class Test                     c)   Class Test
      {                                   {
        Test (int i)                        Test ( int i)
        {                                   {
          Super ();                           Super ();
        }                                   }
      }                                   }
```

** Super & This :-

→ The first Line inside a Constructor should be either Super()(or) this().

→ If we are not writing any thing Compiler will always places Super().

Case (i) :-

we have to keep either Super()(or) this() only as the first Line of the Constructor.

```
class Test
{
  Test()
  {
    S.o.p(" Hi ");
    Super();   ✗  →  C.E:- Call to Super must be first
                          Statement in Constructor.
  }
}
```

Case (ii) :-

with in the Constructor we can use either super() or this() but not both Simultaniously.

```
Class Test
{
  Test()
  {
    Super(); ✓
    this(); ✗  →  C.E:-  Call to this must be first statement
                          in the Constructor
  }
}
```

Case (iii) :-

→ we can use super & this only inside constructor if we are using any where else we will get compile time error.

Eg:

```
class Test
{
  public void m1()
  {
    Super();   X  →   C.E:- Call to Super must be
                            first statement in the constructor
    S.o.pln("Hi");
  }
}
```

Super() → must be used only in constructor
this() → as the first statement only
→ But not both simultaniously.

this() :- To call current class constructors

Super() :- To call parent class constructors

Compiler provides default super() but not this().

Super()
this()
	Super
	this

(1) these are constructor calls	1) these are keywords to reffere. Super & current class instance members
(2) we should use only in constructors	2) we can use any where except in static area.

Ex:-

```
Class Test
{
 p.s.v.m()
 {
  S.o.pln( Super.hashCode()); X
 }          ↳ CE:- NON-Static variable Super can't be
                   referenced from a Static Context
}
```

Constructor overloading:-

→ A class can contain more than one constructor with same name but with different arguements & these constructors are considers as overloaded constructors.

ex:-

```
Class Test
{
 Test(double d)
 {
  this(10);
  S.o.pln(" double-args");
 }
 Test(int i)
 {
  this();
  S.o.pln(" int-args");
 }
Test()
{
 S.o.p(" No-args");
}
p.s.v.m(———)
{
```

```
Test t1 = new Test(10.5);  →  No-args
                              int-args
                              double-args

Test t2 = new Test(10)  →  No-args
                           int-args

Test t3 = new Test()  →  No-args
```

**→ Inheritance & overriding concepts are not applicable for Constructors.

**→ Every class in java including abstract class also can contain Constructor. But interfaces can't have the constructors.

```
Class Test         | abstract class Test  | interface Test
{                  | {                    | {
  Test()           |   Test()             |   Test()
  {                |   {                  |   {
  }                |   }                  |   }
}      ✓           | }      ✓             | }      ✗
```

→ Case (i):-

→ Recursive method call is always runtime Exception where as recursive Constructor invocation is a Compile time Error.

ex:-

```
Class Test
{
  p.s.v.m1()
  {
    m2();
  }
  p.s.v.m2()
  {
    m1();
  }
  p.s.v.m(——)
  {
    S.o.p("Hello");
    m1()
  }
}
```

```
|   :    |
| m2()   |
| m1()   |
| m2()   |
| m1()   |
| main() |
```

R.E:. Stack over Flow Error

```
Class Test
{
  Test()
  {
    this(10);
  }
  Test(int i)
  {
    this();
  }
  p.s.v.m(——)
  {
    S.o.pln("Hello");
  }
}
```

C.E:- Recursive Constructor invocation.

Case (ii) :-

Ex:-

```
Class P
{
}
Class C extends P
{
}
```

✓

```
class P
{
  P()
  {
  }
}
Class C extends P
{
}
```

✓

```
Class P
{
  P(int i)
  {
  }
}
Class C extends P
{
}
```

✗

```
C()
{
super();
}
```

C.E:-

Can't find Symbol

Symbol : Constructor P()

location : Class P.

Note:-

→ if the parent class Contains Some Constructors then while writing child class we have to take Special Care about Constructors.

→ when ever we are writing any arguement Constructor it is highely recommended to write no arguement Constructor also.

Case (iii) :-

→ if parent class constructor throws Some Checked Exception Compulsary Child class constructor should throw Same checked Exception or its parent other wise the Code won't Compile.

```
class P
{
  P() throws IOException
  {
  }
}
```

```
Class C extends P
{
}
```

C.E:- unreported Exception Java.io.IOException in default Constructor

```
Ex.-   Class P
        {
          P() throws IOException
           {
           }
        }

        Class C extends P
         {
            C() throws IOException/Exception
            {
            }
         }
```

Q) which of the following is True?

① Every class Contains Constructors ✓

② only Concreate classes Can Contain Constructors but not abstract Classes. ✗

③ the name of the Constructor need not be same as class name ✗

④ returntype is applicable for the Constructor ✗

⑤ The only applicable modifiers for Constructors are public & default ✗

⑥ If we are trying to declare returntype for the Constructor we will get Compiletime Error ✗.

⑦ Compiler will always generate default Constructor ✗

⑧ The access modifier ~~of~~ the default Constructor is always default ✗.

⑨ The first Line inside every Constructor should be Super ✗.

⑩ The first Line " " should be Super() or this() ✓

if we are not writing any thing compiler will always place this() ✗.

(11) Interface Can Contains Constructor ✗

(12) Both overloading & overriding Concepts are applicable for Constructor ✗.

(13) Inheritance Concept is applicable for Constructor ✗

Type-Casting

Type-Casting:-

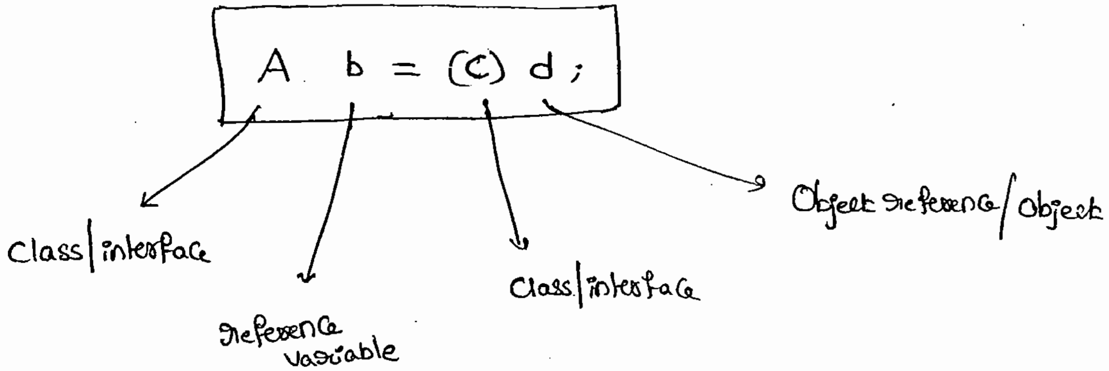
→ Parent class reference Can be used to hold child class object

Ex:- Parent p = new child();

→ Similarly, interface reference Can be used to hold implemented class object.

ex: Runnable r = new Thread();

Syntax:-



Compiler rule(1):

→ C & type of d must have Some relationship (either parent to child or child → parent or Same type) otherwise we will get Compiletime Error Saying "inconvertable types found d type but required C type".

eg(1) :-

```
Object o = new String("durga");

StringBuffer sb = (StringBuffer) o;
```

eg(2) :-

```
String s = new String("durga");

    SB sb = (SB)s;        ✗
```

C.E:- inconvertable types
found :- java.lang.String
required : java.lang.SB

Compilerchecking rule 2 :-

→ C must be either same or derived type of A otherwise we will get compile time error saying "incompatable types"
found : C
required: A

Ex(1):-

```
Object o = new String("durga");

String s = (String) o;     ✓
```

Ex(2):-

```
String s = new String("durga");

StringBuffer sb = (Object)s;
```

C.E:- incompatable types
found : Object
required : SB

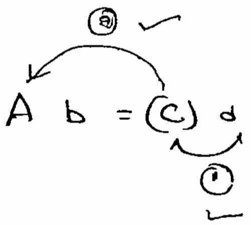
Runtime checking
Rule 3 :-

→ The underleying object type of 'd' must be either Same or derived type of C, otherwise we will get runtime Exception Saying "ClassCastException".

Ex:- ① Object o = new String("durga");

SB sb = (SB) o; ✗

Rule ① ✓
② ✓
③ ✗ (R.E):- CCE

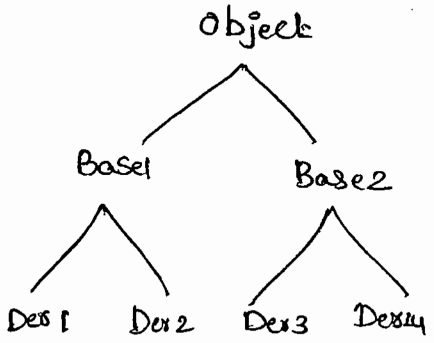


② Object o = new String("durga");

String s = (String)o; ✓

Rule ① ✓
② ✓
③ ✓

Ex:-

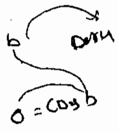


Ex ① Base2 b = new Der4(); ✓

✓ ② Object o = (Base2) b;

✗ ③ Object o = (Base1) b;

④ Base2 b1 = (Base2) o;



✗ ⑤ Base1 b3 = (Der1)(new Der2());

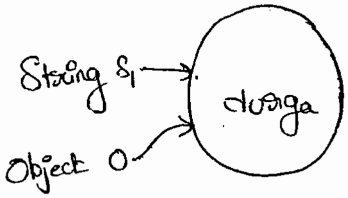
C.E:- inconvertiable types
found : Base2
required : Base1

C.E:-
inConvertiable types
found: Der2
required : Der1

→ Strictly Speaking in type-Casting just we are Converting only type of object but not underlying object itSelf

Eg:-

```
String s1 = new String("durga");

Object o = (object) s1;

S.o.pln(s1 == o);  true
```



Ex:-

```
A ────→ public void m1()
↑        {
│         S.o.pln("A");
│        }
B ────→ public void m1()
↑        {
│         S.o.pln("B");
│        }
C ────→ public void m1()
         {
          S.o.pln("c");
         }
```

```
C c = new C();
      c.m1();   o/p c ✓
((B)c).m1();   → c ✓ ──→ B b = new C();
                           b.m1();
((A)c).m1();   ──→ c ✓
                    ──→ A a = new C();
                          a.m1();
```

Eg2:-

```
A ⟶ public static void m1()
↑   {
|    S.o.pln("A");
|   }
|
B ⟶ public static void m1()
↑   {
|    S.o.pln("B");
|   }
C
  ⟶ public static void m1()
    {
     S.o.pln("C");
    }
```

*) C c = new C();

c.m1(); // C ✓

*) ((B)c).m1(); // B ✓

*) ((A)c).m1(); // A ✓

Eg3:-

```
A ⟶ int x = 777;
↑
B ⟶ int x = 888;
↑
C ⟶ int x = 999;
```

C c = new C();

S.o.pln(c.x); 999 ✓

S.o.pln(((B)c).x); 888 ✓

S.o.pln(((A)((B)c)).x); 777 ✓

(because the overriding Concept is not applicable for variable).

→ if we declare all variables as static then there is no chance to change the o/p.

Note:-

→ wheather the variable is static or instance variable resolution should be done based on reference type but not based on runtime -object.

Coupling

Coupling:-

→ The degree of dependency b/w the components is called "coupling"

Ex:-

```
Class A
{
  static int i = B.j;
}
```

```
class B
{
static int j = C.m1();
}
```

```
class C
{
 p.s.v.int m1()
 {
  return D.k;
 }
}
```

```
class D
{
static int k =10;
}
```

→ The above components are said to be tightly coupled with each other. Tightly coupling is not recommended because it has several serious disadvantages.

(1) with out effecting ~~any~~ remaining component we can't modify any components

Hence, enhancement will become defficault.

2, it reduces maintainability.

3, It doesn't promote reusability.

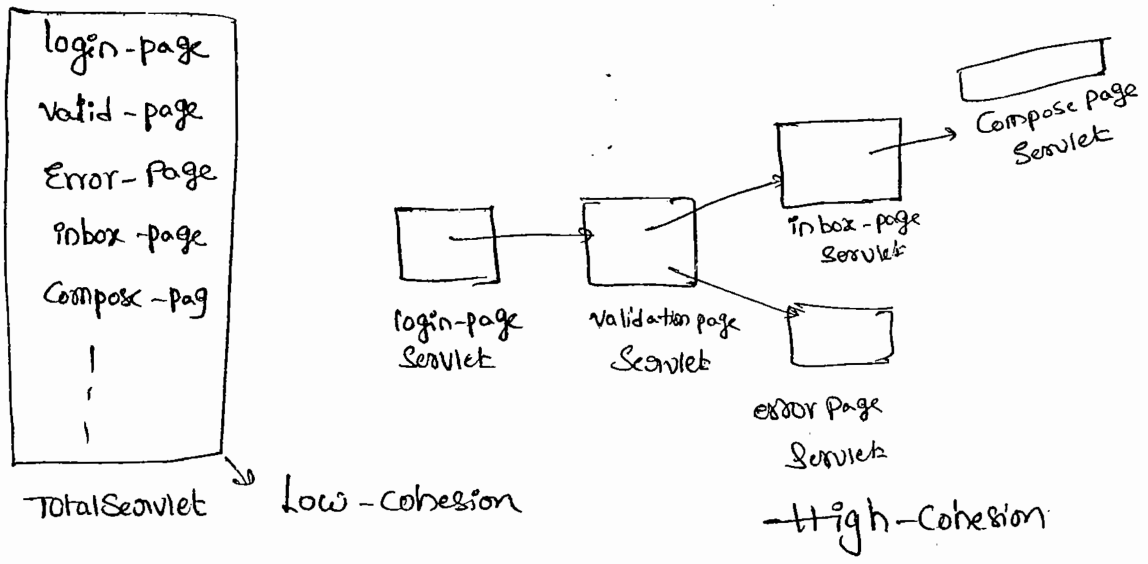
→ Hence it is highly recommended to maintain loosely coupling & dependency b/w the Components should be as less as possible.

Cohesion

Cohesion :-

→ for every Component a clear well-defined functionality we have to define, Such type of Component is Said to be fallow High-cohesion.

Ex:-



→ High-Cohesion is always a good programing practice which has Several Advantages.

(1) with out effecting remaining Components we can modify any Component Hence enchancement will become Very easy

(2) It improves maintainability of the application

(3) It promotes reuseability of the code.

Ex:-

→ where ever validation is required we can reuse the same Validate Servlet without rewriting.

Note:-

Loosely Coopling & high-Cohesion are good programing practices.

═ xox ═

This :

to use the current class reference

means with out Creating mutiple objects, only one object is Created then Call those values from the Current class.

Kondalu_7@yahoo.co.in

మా, వే

I would try to update our site JavaEra.com everyday with various interesting facts, scenarios and interview questions. Keep visiting regularly.....

Thanks and I wish all the readers all the best in the interviews.

www.JavaEra.com

A Perfect Place for All Java Resources