

Durgasoft SCJP Notes

Part-1

Java

Language Fundamentals

MANOJ PUNJABI B.Sc. (Hons) & M.Sc.
I PLOT KALYAN CAVALRY NAGAR
AMRITSAR PUNJAB INDIA 148102

- ① Identifiers 1
- ② Reserved Words 2
- ③ Data types 5
- ④ Literals 9
- ⑤ Arrays 13
- ⑥ Types of Variables 22
- *⑦ var-arg methods 23 (1.5 version)
- ⑧ main() method 30
- ⑨ Command-line arguments 33
- ⑩ Java Coding Standards 34

1) Identifier :-

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

Ex:- `Class Test` → class name
↓
`p.s.v. main(String [] args)` → method name
↓
`int x=10;` → is identifier.
↓
} → variable name

* Rules to define identifiers:-

1) The only allowed characters in Java identifier are:

✓ $\left(\begin{array}{l} a \text{ to } z \\ A \text{ to } Z \\ 0 \text{ to } 9 \\ - \\ \$ \end{array} \right)$

→ If we are using any other character we will get Compilation Error.

Ex:-

✓ all-member

✗ all#

✓ -\$-\$

✗ 098\$-10

2) Identifier can't starts with digit. Ex:- ✗ 123total

✓ total123.

3). Java identifiers are Case Sensitive.

Class Test

{

int Number = 10;

int NUMBER = 20;

int Number = 30;

}

We can differentiate w.r.t Case.

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

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

6) All predefined Java class names & interface names we can use as
identifiers. ~~but~~ Even though it is legal, but it is not recommended.

Ex:-

Class Test

{

int String = 10;

S.o.println(String); 10

}

Class Test

{

int Runnable = 20;

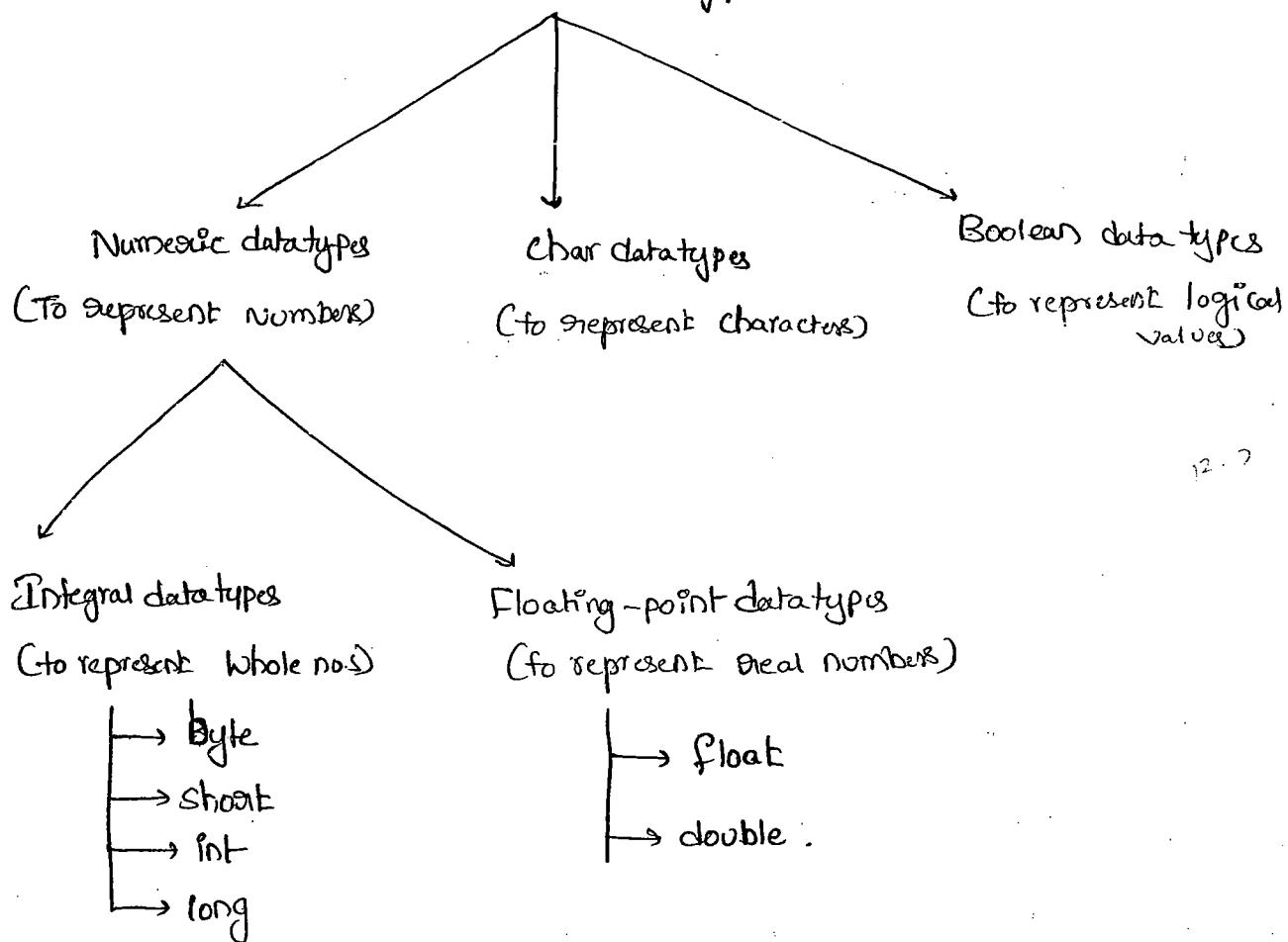
S.o.println(Runnable); 20

}

Q) Which of the following are valid Java identifiers?

- ① Java\$share
- ② 4shared
- ③ all@hands
- ④ total-not-Students
- ⑤ -\$-
- ⑥ total#
- ⑦ int
- ⑧ Integer

Primitive data types (8)



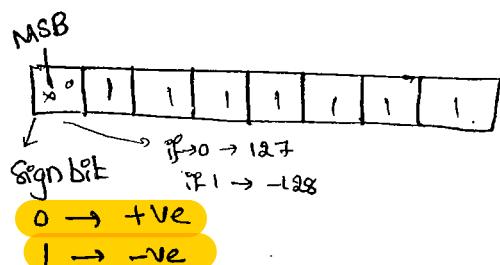
① Byte :-

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 whereas -ve numbers

represented in 2's Complement form.

Ex:-

byte b = 100;

✓ byte b = 127;

X byte b = 130; C.E:- possible loss of precision

found : int

Required : byte

X byte b = 123.456; C.E:- PLP

found : double

Required : byte

X byte b = true; C.E:- PLP incompatible types

found : boolean

Required : byte

X byte b = "durga"; C.E:- incompatible types

found : 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 $2^{15}-1$,

$[-32768 \text{ to } 32767]$

Ex! ✓ Short s = 32767

✓ Short s = -32768

X 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 frequently used datatype in Java is short

→ Short datatype is best suitable if we are using 16-bit processors

like 8086 but these processors are Completely outdated & hence

Corresponding short datatype is also **outdated**.

(3) int :-

→ The most **Commonly** used datatype is int

Size : 4-bytes

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

$[-2147483648 \text{ to } 2147483647]$

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

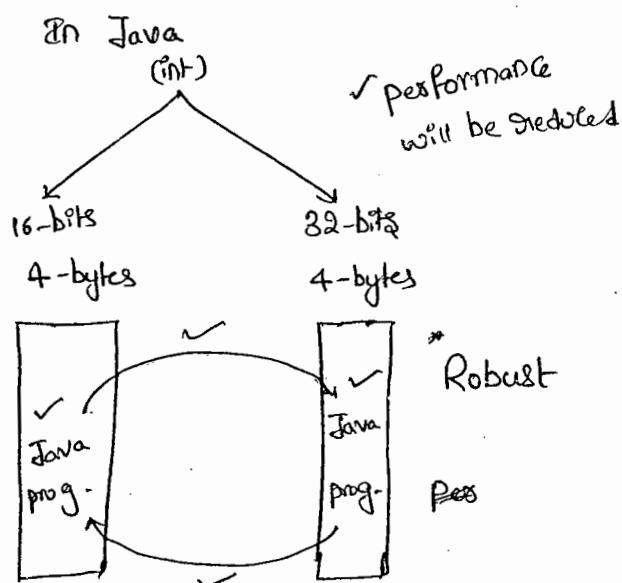
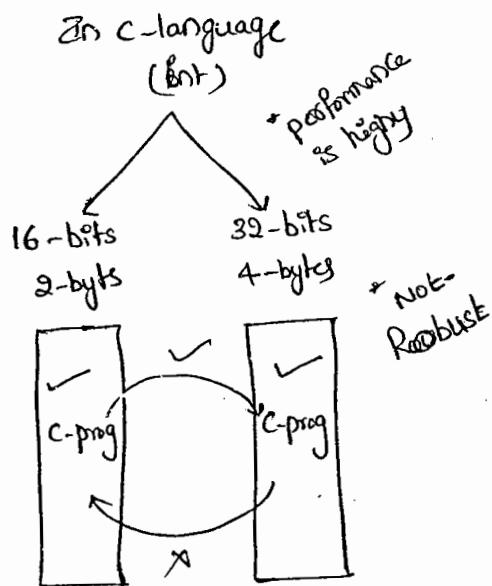
Very efficiently and performance will be improved. But the main

* disadvantage of this approach is the chance of **failing** writing C program

is very very high if we are changing platform <http://javabynataraj.blogspot.com> page 255.

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 failing Java program is very very less, if we are changing underlying platform hence Java is considered as Robust language.
- * But the main disadvantage in this approach is read & write operations will become costly & performance will be reduced.



3/08/11

4) long :-

→ When even 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 Compulsory we should go for long type

$$\text{Ex(1)} \rightarrow \text{long } l = 1,23,000 \times 60 \times 60 \times 24 \times 1000 \text{ miles}$$

(2)

Ex(2) :-

To Count the no. of characters present in a big file. int may not enough Compulsory 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) meant for representing whole values.
- If we want to represent real numbers Compulsory we should go for floating point data-types.

Floating Point data-types :-

floating point data-types



1) Size : 4-bytes

2) Range : -3.4×10^{-38} to 3.4×10^{38}

3) If we want 5 to 6 decimal places of accuracy then we should go for float

4) float follows Single precision

1) Size : 8-bytes

2) Range : -1.7×10^{-308} to 1.7×10^{308}

3) If we want 14 to 15 decimal places of accuracy then we should go for double.

4) double follows 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:- Incompatible 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

↳ o.println(b); true

Ep :-

int x=0;

if(x)

in Java X

{
System.out.println("Hello");

} else

{
System.out.println("Hi");

C++ ✓

C.E:- incompatible types

→ found : int

Required : boolean

in Java X
while(1)

{
System.out.println("Hello");

in C++ ✓

→ 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 alphabets sets. The no. of unicode characters is >256 & hence 1-byte is not enough to represent all characters Compulsory We should go for 2-bytes.

Size : 2-bytes

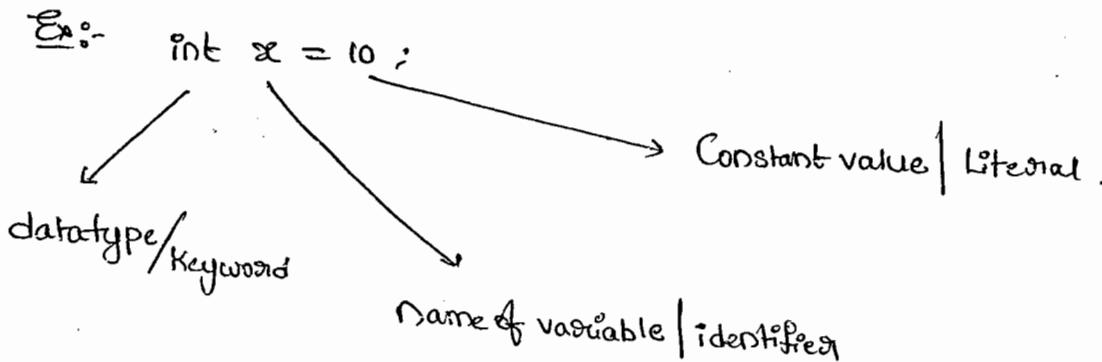
Range : 0 to 65535

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/false)

Literals :-

→ A Constant value which can be assign to the Variable is called "Literal"



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

8

Exo- `int x = 0x10`

(?)

`int x = 0X10`

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

Exo- class Test

{

p.s.v.m (String [] args)

{

`int x = 10;`

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

`int y = 010;`

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

`int z = 0X10;`

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

`S-o.println(x + "----" + y + "-----" + z);`

$$10 \quad \quad \quad 8 \quad \quad \quad 16 \quad \quad \quad 0 \times 16 + 1 \times 16^1 = 16$$

}

QUESTION

Q) Which of the following declarations are valid.

✓ ① `int x = 10;`

✓ ② `int x = 066;`

X ③ `int x = 0786;` C.E: integer number too large

✓ ④ `int x = 0xFACE;` 64206

X ⑤ `int x = 0XBEEF;` C.E: (after B) ; Excepted

✓ ⑥ `int x = 0xB6a;` 3050

→ By default 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

✓ 3) long l = 10L; found: long
Required: int

✓ 4) long l = 10;

→ There is no way to specify integral literal is to byte & short types explicitly.

→ If we are assigning integral literal to the byte variable & that integral literal is within 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 by default 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

✓ float f = 123.456f;
✓ double d = 123.456;

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

Ex:- ✓ double d = 123.4567D;

X float f = 123.4567d; c.e.: P.L.P

→ 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 = 0xBear;

✓ 6) float f = 0.642; // 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 datatype.

& 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:- ~~X int i = 123.456;~~ **PLP**

→ found : double
Required : int

✓ ~~double d = 1.2e3;~~

~~S.o.println(d); 1200.0~~

→ we can specify floating point literal even in scientific form
also [exponential form]

Ex:- ✓ 1) ~~double d = 1.2e3;~~

~~S.o.println(d); 1200.0~~

✓ 2) ~~float f = 1.2e3; C.E.: PLP~~

→ found : double

✓ 3) ~~float f = 1.2e3f; o/p:- 1200.0~~

→ required : float

Boolean Literals:

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

Q) Which of the following Boolean declarations are valid?

X ① boolean b = 0; C.E.:- Incompatible types

→ found : int

X ② boolean b = True; C.E.:- Can't find symbol

→ required : boolean

✓ ③ boolean b = true; Symbol : variable True

→ found : boolean

X ④ boolean b = "true"; C.E.:- Incompatible types

→ found : java.lang.String Required : boolean

Ex:- `int x=0;`

```

- if(x)
  {
    S.o.println("Hello");
  }
  else
  {
    S.o.println("Hi");
  }

```

```

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

```

C.E :- Incompatible types

found : int

Required : boolean

Ex@:-

`int x=10;`

```

if(x = 20)
{
  S.o.println("Hello");
}
else
{
  S.o.println("Hi");
}

```

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

`int x=10;`

```

if (x == 20)
{
  S.o.println("Hello");
}
else
{
  S.o.println("Hi");
}

```

O/P: Hi

`boolean b=true;`

```

if(b=false)
{
  S.o.println("Hello");
}
else
{
  S.o.println("Hi");
}

```

O/P:- Hi

`boolean b=true;`

```

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

```

O/P:- Hello

Char Literals :-

→ A char literal can be represented as single character with in single quotes.

Ex:- ✓ char ch = 'a';

✗ char ch = a; C.E:- Can't find symbol

Symbol: Variable a

✗ char ch = 'ab'; location : class xxxx

→ C.E:- unclosed character literal

C.E:- unclosed "

C.E:- not a statement

25/08/11

→ 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:- ✓) char ch = 97;

S.o.p(ch); a

✓ 2) char ch = 65535;

S.o.println(ch);

✗ 3) char ch = 65536; C.E:- PLP

-found: int

Required: char

✓ 4) char ch = 0XFACE;

✓ 5) char ch = 0640;

3) A char literal can be represented in Unicode representation
which is nothing but $\boxed{\backslash Uxxxx}$ 4-digit hexa decimal no.

Ex:- 1) char ch = '\u0061';

S.o.p(ch); a

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

✓ 3) char ch = '\uface';

X 4) char ch = '\i beaf';

4) Every escape character is a char literal

Ex:- 1) char ch = '\n';

✓ 2) char ch = '\t';

X 3) char ch = '\l';

Escape Character	meaning
\n	New Line
\t	Horizontal tab
\r	Carriage Return
\b	Back Space
\f	form feed
'	Single quote
"	Double quote
\	Back slash

Q) Which of the following are valid char declarations.

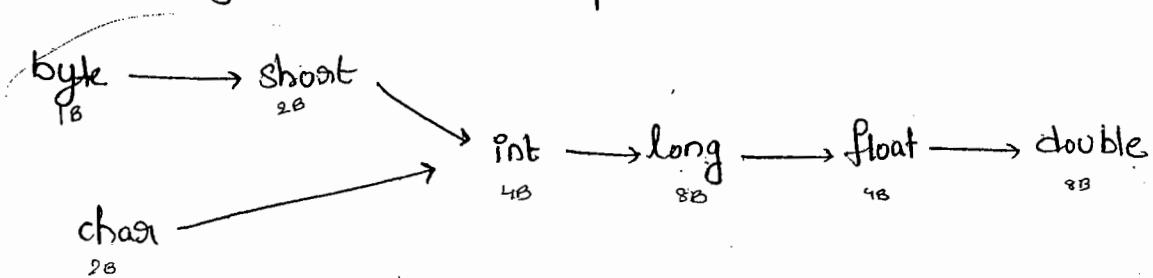
- ✓ 1) char ch = 0xbeaf;
- ✗ 2) char ch = \Ubeaf; because ' '
- ✗ 3) char ch = -10;
- ✗ 4) char ch = '*';
- ✓ 5) char ch = 'a';

String Literals :-

→ Any Sequence of characters with in " " (double quotes) 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. Anonymous 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 is 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 is recommended because Type is clearly separated from the 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;` $a \rightarrow 1$
 $b \rightarrow 1$ 

2) `int[] a[],b;` $a \rightarrow 2$
 $b \rightarrow 1$

3) `int[] [] a, b;` $a \rightarrow 2$
 $b \rightarrow 2$

4) `int[] [] a, b[];` $a \rightarrow 2$
 $b \rightarrow 3$

X 5) `int[] [] a, [] b;` $a \rightarrow 2$
 $b \rightarrow 3$ C.E. :-

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

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

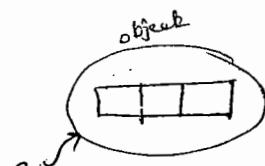
Allowed

not allowed;

Q) 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. These classes are not applicable for programmer level.

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

→ At the time of Construction Compulsory we should Specify the size otherwise we will get C.E..

Ex:- `int[] a = new int[];` X 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
<http://javabynataraj.blogspot.com>

→ 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;

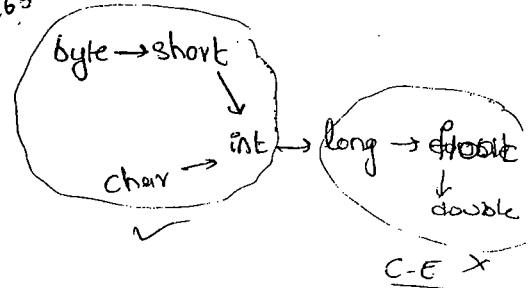
✓ `int[] b = new int[b];`

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.

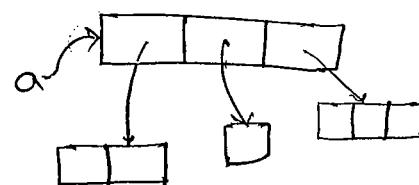
→ The **main advantage** of this approach is **memory utilization** will be improved.

Ex:- `int[][] a = new int[3][];`

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

`a[1] = new int[1];`

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



Note:-

In C++, as

Ex 2:

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

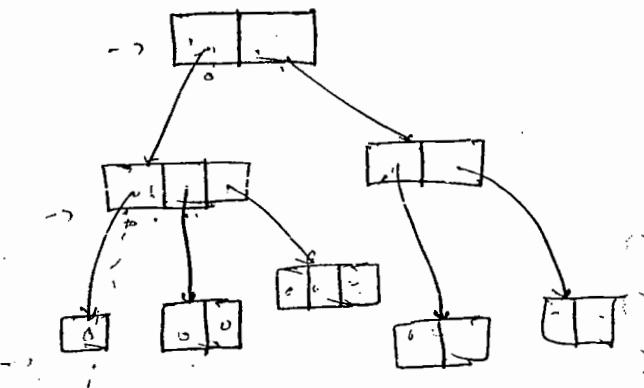
`a[0] = new int[3];`

`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];`

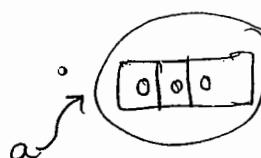
Array Initialization :-

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

Ex(1): `int[] a = new int[3];`

`S.o.println(a);` [I@3e25a5 hashcode]

`S.o.println(a[0]);` 0



Note:-

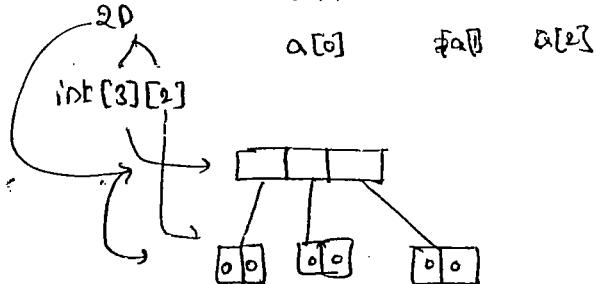
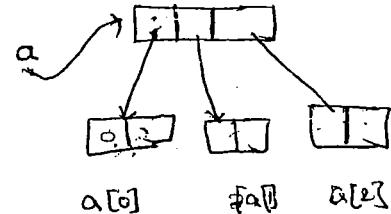
Whenever we are trying to print any object reference internally

`toString()` will be called which is implemented as follows.

classname @ hexadecimal_string_of_hexcode.

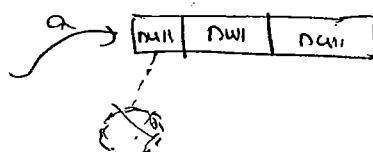
Ex(2):-

```
int[][] a = new int[3][2];
System.out.println(a); [[I@-----
System.out.println(a[0]); [I@ 4567
System.out.println(a[0][0]); 0.
```



Ex(3):-

```
int[][] a = new int[3][];
System.out.println(a); [[I@-----
System.out.println(a[0]); null
System.out.println(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;
```

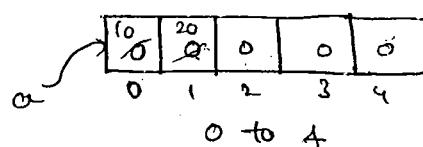
```
a[2] = 40;
```

```
a[3] = 50;
```

```
a[4] = 60;
```

```
a[-50] = 30;
```

```
a[10.5] = 30;
```



R.E: AIOBE

R.E: AIOBE

Note:-

C.E:- PLC, found = double, required = int.

→ 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[4];
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 = {"Sneha", "Ravi", "Daxmi", "Sundar"};

→ we Can Extend This Shortcut Even for multidimensional arrays also.

Ex(3):-

```

int[][] a = {{30, 40, 50}, {60, 70}};
            ↙   ↘
            a[0]  a[1]
            ↙   ↘
            30 40 50   60 70
            ↙   ↘
            a[0][0] a[1][0]
    
```

→ we Can Extend This Shortcut Even for 3D array also

Ex:-

```

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

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

`S.o.println(a[1][2][3]);` RE:- ALOBE

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

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

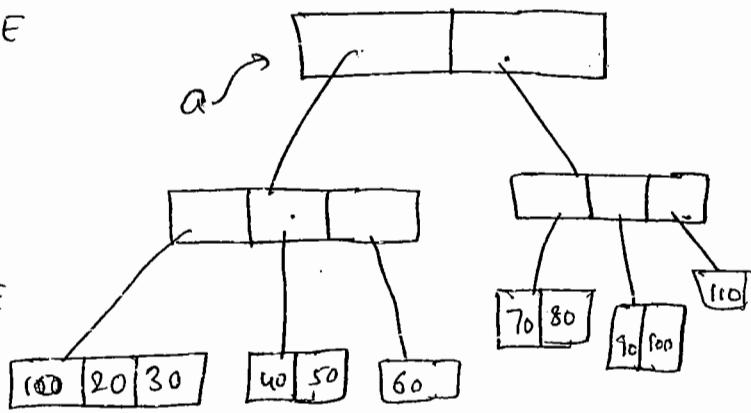
`S.o.println(a[3][1][2]);` RE:- ALOBE

`S.o.println(a[2][2][2]);` RE:- ALOBE

`S.o.println(a[1][1][1]);` 100

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

`S.o.println(a[1][0][2]);` RE:- ALOBE



→ If we want to use Shortcut Compulsory we should perform

declaration, Construction & initialization in a Single Line.

→ 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 = {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.println(a.length); 10`

`S.o.println(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.println(s.length()); 5`

`S.o.println(s.length());`

↳ E! Cannot find Symbol
Symbol: variable length
location: java.lang.String.

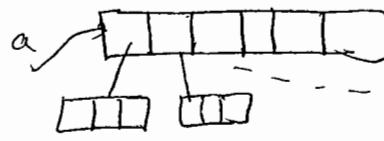
→ In multidimensional arrays length variable represents only

base size, but not total size.

Eg:- `int[][] a = new int[6][3];`

`S.o.println(a.length); 6`

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



Note:-

- lengths variable is applicable only for arrays whereas length() is applicable for String objects.

Anonymous Array :-

- Sometimes we can create an array without name also

Such type of nameless arrays are called "Anonymous arrays".

- The main objective of anonymous array is **Just for instant use.**
(not future) **(only once)**

- We can create Anonymous array as follows.

`New int[]{10, 20, 30, 40}`

- At the time of Anonymous Array Creation we can't specify the size, otherwise we will get Compilation 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 i : x)
        {
            total = total + i;
        }
        System.out.println("the Sum : " + total); 100
    }
}

```

→ Based on our requirement we can give the name for Anonymous array, then it is no longer Anonymous.

Eg:-

```

String[] s = new String[]{"A", "B"};
    ↗ S.out(s[0]); A
    ↗ S.out(s[i]); B
    ↗ S.out(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.

Q) 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 Compiletime Error.

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: float

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

byte → short

int → long → float → double

char

Case(2):-

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

Eg:-

① Number[] n = new Number[10];

✓ n[0] = new Integer(10);

✓ n[1] = new Double(10.5);

✗ n[2] = new String("doung"); → C.E:- Incompatible types

found: String

Required: Number

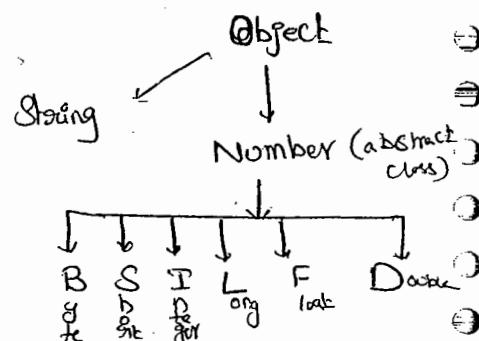
② Object[] a = new Object[10];

✓ a[0] = new Object();

✓ a[1] = new Integer(10);

✓ a[2] = new Double(10.5);

✓ a[3] = new String("deaga");



Case(3):-

→ In the Case of abstract class type arrays as array elements we can provide its child class Objects.

Eg:- 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

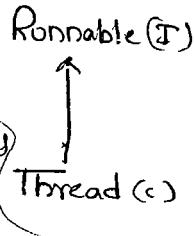
Can provide it's implementation class Objects

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

r[0] = new Thread();

X r[1] = new String("doings"); C.E! - Incompatible type

→ 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 implementation class Objects are allowed

Array Variable Assignment :-

Case(1) :-

→ Element level promotions are not applicable at array level

Eg:- A char value can be promoted to int 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 = cb; C.E:- Incompatible 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 → Object^(Parent)
(Child)

✓ ⑧ String[] → Object[]

Eg:- Child-type array, we can assign to the parent-type variable.
<http://javabynataraj.blogspot.com> 37 of 255.

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

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

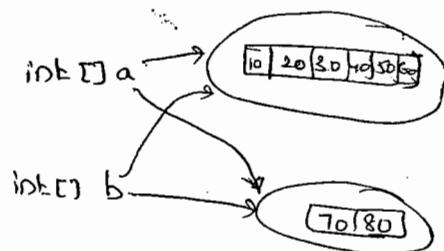
✓ Object() a = s;

Ques(2)?

→ When even we are assigning one array to another array only reference variables will be reassigned but not underlying elements.
Hence types must be matched but not sizes.

Eg:- Ex:- 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];

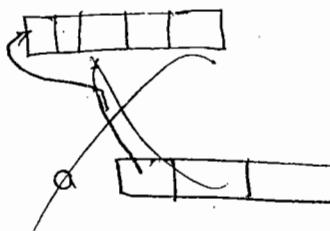
a[1] = new int[4];

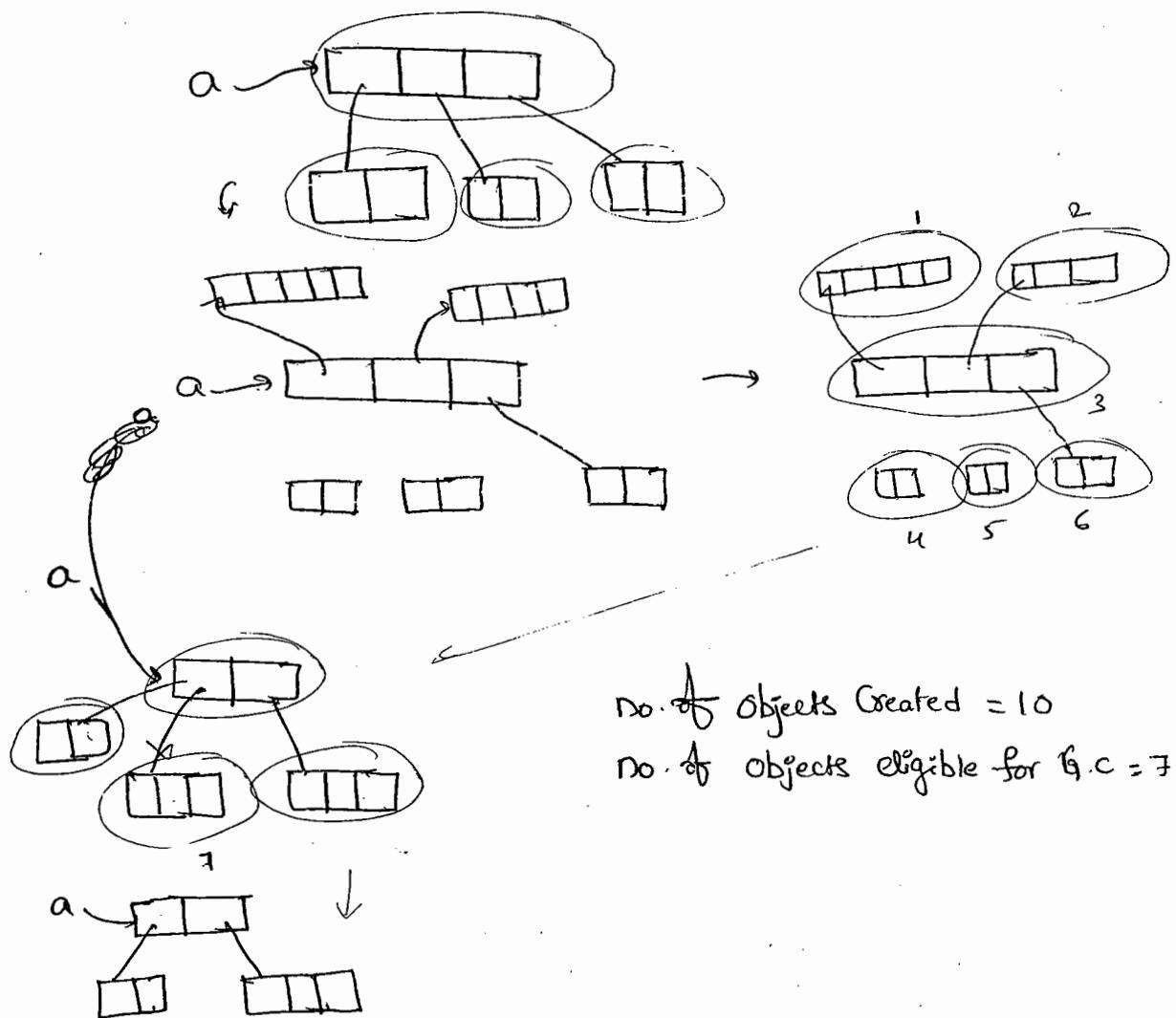
a = new int[2][3];

a[0] = new int[2];

No. of objects created ≈ 10

No. of objects eligible for G.C ≈ 7 .





Case 3:-

→ Whenever we are performing array assignments dimensions must be matched, i.e. in the place of Single dimensional int[] array, ~~only~~ we should provide only Single dimensional int].
by mistake we are providing any other dimension we will get Compilation Error.

`Eqo- m[0][0] = new m[3][];`

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

$$a[0] = \text{new } \partial t[3][2]; \rightarrow$$

QГРЛ-20:

C. E : incompatible types
→ found : int C())
<http://javabynaturel.blogspot.com>

`a[0] = 10; C.E!:` Incompatible types
 found : int
 Required : int[]

22

Types of Variables

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

(i) primitive Variables

(ii) Reference Variables

(i) Primitive Variables

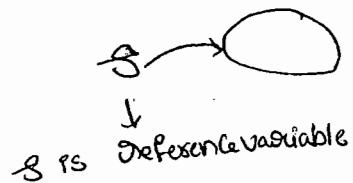
→ Can be used to represent primitive values

Ex:- `int x = 10;`

(ii) Reference Variables

→ Can be used to refer Objects

Ex:- `Student s = new Student();`



→ 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 Separate 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 blocks or Constructor.

→ Instance variables Cannot be accessed from 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.P.N(x); → C.E:- non-static variable x Cannot

be referenced from static context

Test t = new Test();

23

s.o.println(t.x); so ↗

```
}  
public void m() {  
    }  
    s.o.println(); ✓ ↗  
}
```

→ 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();

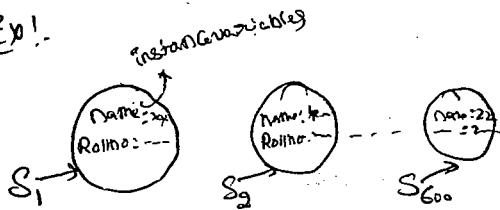
s.o.println(t.s); null

s.o.println(t.x); 0

s.o.println(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;
```

```
,
```

```
,
```

```
,
```

```
{
```

```
    name: xxx  
    Rollno: 101  
    Cgpa: 3.5
```

```
S1
```

```
name:yyy  
Rollno: 202
```

```
S2
```

```
name:zzz  
Rollno: 600
```

```
S600
```

College-name: duagastu

→ 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 separate 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** & destroyed at the time of **Class unloading**. Hence the scope of the static variable is

Exactly Same as the Scope of the class.

gfp

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 blocks 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.println(Test.x); ✓ 10

S.o.println(x); ✓ 10

✓ Test t = new Test();

↳ S.o.println(t.x); ✓ 10

↳ S.o.println(x); ✓ 10

→ Static variables are created at the time of class loading i.e., (at the beginning 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.println(x);
        }
        public void m1()
        {
            s.o.println(x);
        }
    }
```

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

Eg:- Class Test

```
    {
        static int x;
        p.s.v.m(String[] args)
        {
            s.o.println(x);
        }
    }
```

→ 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();

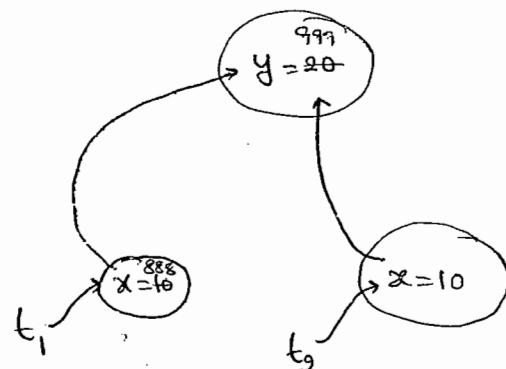
 S.O.Pln(t₂.x+"---"+t₂.y);

 }

 t₀

999

}



t₁.x=888

t₂.x=10

t₁.y=999

t₂.y=20

y=999

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

(ii) Local Variables:-

- To meet temporary requirements of the programmer sometimes we have to create variables inside method or Block or Constructor. Such type of variables are called Local variables.
- Local variables also known as **Stack variables** or **Automatic variables** or temporary variables.
- Local variables will be stored inside a **Stack**.
- The Local variables will be created while executing the block in which we declared it & destroyed once the Block Completed. Hence, the **Scope of Local Variable** is exactly 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. p( 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.pn("Hello");

}

%P:- Hello

Class Test

↓

p.s.v.m(String[] args)

↓

int x;

S.o.pn(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.pn(x);

}

C.E:- Variable x might not have been initialized
<http://javabynataraj.blogspot.com> 48 of 255.

Eg 3: Class Test

```
    {
        p.s.v.m(String[] args)
    }

    int x;

    if(args.length > 0)
    {
        x = 10;
    }
    else
    {
        x = 20;
    }

    S.o.println(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 guarantee 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)

}

X private int x=10;

X public int x=10;

X protected int x=10;

X static int x=10;

✓ final int x=10;

}

}

C.E:-

Illegal Start of Expression.

Uninitialized Arrays..

Class Test

{

int[3] a;

P.S.V.m (String[] args)

}

Test t, = new Test();

S.o.println(t.a); null

S.o.println(t.a[0]); Nonpointer Exception

}

instance level:-

int [] a ;

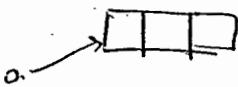
i.e. a=null

S.o.p(obj.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) difference 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 below 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 been initialized
S.o.p(a[0]) }

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 whether it is static or instance or local array.

↳ Var-arg methods (1.5 version)

→ Until 1.4 version we can't declare a method with variable no. of arguments, if there is any change in no. of assignments Compulsory 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 arguments. Such type of methods are called Var-arg methods.

→ We can declare Var-arg method as follows.

`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(i) :-

Class Test

```

    {
        p.s.void m1(int... i)
        {
            s.o.println("Var-arg method");
        }
        p.s.v.m(String[] args)
        {
            m1();
            m2();
            m3();
            m4();
        }
    }
  
```

Op:- Var-arg method
 Var-arg method
 $\begin{matrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{matrix}$

→ Internally var-arg method is implemented by using single dimension arrays concept. Hence within the var-arg method we can differentiate arguments by using index.

Ex:- Class Test

```
↓  
public static void sum(int... x)  
↓  
int total = 0;  
for(int y: x)  
{  
    total = total + y;  
}  
System.out.println("The Sum: " + total);  
↓  
D.S.V.M(String[] args)  
↓  
sum(); 0  
sum(10, 20); 30  
sum(10, 20, 30) 60  
sum(10, 20, 30, 40); 100  
↓  
}
```

Op! 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...) ✗

m1(int ...x) ✓

m1(int. ...x) ✗

m1(int .x..) ✗

Case 2:-

→ We can mix Var-arg parameters with normal parameters also.

Ex:- m1(int x, String... y) ✓

Case 3:-

→ If we are mixing Var-arg parameters with general parameter

Then Var-arg parameter should be last parameter.

Ex:- m1(int... x, String y) ✗

Case 4:-

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

Ex:- m1(int... x, String... y) ✗

Case 5:- Class Test

p.s.v.m1(int i)

↳ S.o.pIn("General method");

p.s.v.m1(int... i)

↳ S.o.pIn("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.println("int[]");
 }
 P-S-V.m1(int... x)
 {
 S-O.println("int...");
 }
 }

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

Var-arg Vs Single dimensional arrays:-

Case 1:-

→ wherever Single dimensional array present we can replace with var-arg parameter.

Ex:- m1(int[] x) \Rightarrow m1(int... x) ✓

main(String[] args) \Rightarrow main(String... x) ✓

Case 2:-

→ wherever var-arg parameter present we can't replace with Single dimensional array.

~~m1(int... x) \Rightarrow m1(int[] x)~~

09/03/11

30

main()

main()!

- Whether the class contains main() or not & whether the main() is properly declared or not, these checkings are not responsibilities of Compiler. At runtime, **JVM is responsible** for these checkings.
- 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 x 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 anywhere

without existing
object also JVM
has to call this method

main method

can't return
anything to JVM

Command-line
arguments

Name of method
which is configured
inside JVM

→ If we are performing 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 we can 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) \Rightarrow main (String... args)

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

(i) final

(ii) Synchronized

(iii) Staticfp..

Ex:- Class Test

↓

final static Staticfp Synchronized public void main (String... A)

↓

S. o. pln ("Hello everyone");

}

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

- ~~(i)~~ public static int main(String[] args) X
- ~~(ii)~~ static public void Main(String[] args) X
- ~~(iii)~~ public synchronized Strictfp final void main(String[] args) X
- ~~(iv)~~ Public final static void main(String args) X
- ✓ (v) public Strictfp synchronized static void main(String[] args)

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

~~Anse~~ 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)
    {
        System.out.println("ZLU durga S/w");
    }
}
```

class C extends P.

javac p.java ✓

java p

o/p:- ZLU durga S/w

java c

o/p:- ZLU durga S/w

```

Ex 21. class P
{
    p.s.v.m(String[] args)
    {
        System.out.println(" I Love");
    }
}

class C extends P
{
    p.s.v.m(String[] args)
    {
        System.out.println(" durgaSw");
    }
}

```

javac P.java

```

java P
o/p: I love
java C
o/p:- durgaSw.

```

→ 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[] argument method only. The other method we have to call explicitly.

Ex1- class Test
{
 p.s.v.m(String[] args)
 {
 System.out.println(" durgaSw");
 }
 p.s.v.m(Integer args)
 {
 System.out.println(" is good");
 }
}

o/p:- durgaSw.

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 method
present in
String class

If is a
class-
name

Static variable of
type String present
in Test class

Class System

↓

Static PrintStream out;

↓

System.out.println()

↙

→ It is a
class Name
present in
java.lang

→ It is a method
present in
PrintStream class
Static variable of
type printStream
present in System
class

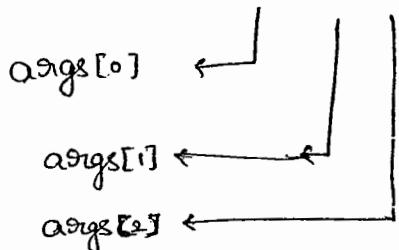
10/03/11

CommandLine Arguments

CommandLine arguments:

- The arguments which are **Passing from Commandprompt** are called **CommandLine arguments**.
- The main objective of **CommandLine arguments** are we can **Customize the behaviour of the main()**.

Ex:- Java Test x y z



args.length ⇒ 3

Ex(1):

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

O/P :- Java Test →

R.E! - AIOBE

Java test x y →

x

y

R.E! - AIOBE

Ex(2):-

→ Within the main(), Commandline arguments are available in String form.

Ex:-

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

Java Test 10 20

O/P:- 1020

- Space is the Separator b/w CommandLine arguments, if the CommandLine arguments itself contain Space then we should enclose with in doubleQuotes ("")

Ex:- class Test

```
{
    p. s. v. m (String[] args)
    {
        s. o. p. n (args[0]); Note Book
    }
}
```

Java Test "Note Book"

Ex(3): class Test

```

    p. s. v. m (String[] args)
    {
        String[] args = {"A", "B"};
        args = args;
        for (String s1 : args)
    }
}
```

s. o. p. n (s1);

```

Java Test x y ←
OR A
B
Java Test x y z ←
OR A
B
Java Test ←
OR A
B

```

Note: The maximum allowed no. of commandline arguments is 2147483647, min. is '0'

Java Coding Standards

→ Whenever we are writing the code it is highly recommended to follow Coding Conventions. 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;
    }
}

```

~~Amescript Standard~~

```

package com.dorgesoft.demo;

public class Calculator
{
    public static int Sum(int number1,
                         int number2)
    {
        return number1+number2;
    }
}

```

Hitech-city

Coding Standards for Classes :-

→ Usually Classnames are Nouns, should starts with Uppercase letter & if it contains multiple words Every inner word should starts with Uppercase letter

Ex:- Student
Customer
String
StringBuffer,

} → NOUNS

→ 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, Closeable, 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 **Lowercase** letter & if it Contains multiple words Every inner words Should Starts with **Upper Case Letter**. (CamelCase).

Ex:-

run()	} → Verbs	} Verb + noun
sleep()		
eat()		
init()		
wait()		
join()		

getName()

setSalary()

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** (CamelCase).

Ex! Name
roll no
mobile Number

} → nouns

⑥ Coding Standards for Constants:-

- Usually The **Constants** are **Nouns**, Should Contain **only Uppercase** characters, If it Contains multiple words, These words are Separated with **" - "** Symbol.
- We Can declare Constants by using **Static & final** modifiers.

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

⑦ Java bean Coding Standards

- A Java **bean** is a Simple java class with private properties & Public getters & Setters 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 official convention
from SUN.

Syntax for Setter method :-

- The method name should be prefix with "Set". Compulsory the method should take some argument. Return type should be void.

Syntax for getter method :-

- The method name should be prefixed with "get".
- It should be no argument method.
- Return type 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;
}
```

Coding Standards for Listener :-

To Register a Listener :-

- Method name should be prefix with add.
- After add whatever we are taking the argument 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 deleteMyActionListener(MyActionListener l)
 - ✗ ④ public void removeMyActionListener(ActionListener l)

Note:-

In Java bean Coding Standards & Listener Concept 1 compulsary.

Operators & Assignments

Kathy Sierra 1-6

book for SCJP

Increment / Decrement 2

Arithmetic operators 3

Concatenation 5

Relational operators 5

Equality operators 6

Bitwise operators 7

Short-circuit 9

instanceof 6

typeCast Operator 10

Assignment Operators 12

Conditional Operator 13

New Operator 13

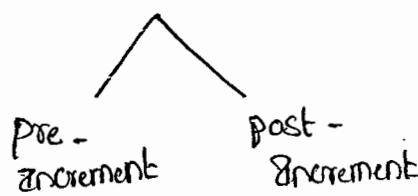
[] Operator 13

Operator precedence 14

Evaluation Order of Java operands. 14

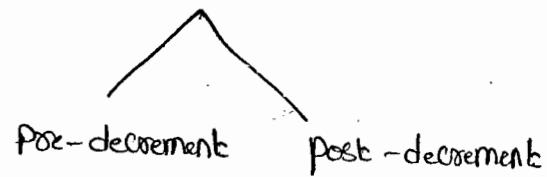
Encryption & Decryption Operations.

Engagement



`int x = +iy;` `int x = y++;`

Decarment



`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

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

int x= 4;

~~X~~ int y = ++4;
Sopln(y); C.E: unexpected type
 ↳ 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;`

`X int y = ++(++x);`

`S.o.p(y);`

C.E: UNExpected type
② found : value
① Required : Variable

→ after scope - y is
- Constant
Then

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

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

Ex(2):- `final int x = 4;` 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;`

`d++;`

`S.o.p(d); // 11.5`

② `char ch = 'a';`

`ch++;`

`S.o.p(ch); // b`

③ `boolean b = true;`

`X`

`++b;`

`S.o.p(b);`

C.E:-

operator ++ can't applied to boolean.

④ `int x = 10;`

`x++;`

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

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

① byte b = 10;
 b++;
 S.o.p(b); //

② byte b = 10
 X b = b + 1;
 S.o.p(b);

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

③ byte b = 10
 b = (byte) (b+1)
 S.o.p(b); //

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

④ byte a = 10;
byte b = 20;
byte c = a+b;
S.o.p(c); C.E: PLP
f = int
R = byte

Explanation:-

Max(int, type of a, type of 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,

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

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

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

byte b++; \Rightarrow $b = (\text{byte})(b+1);$

b++; \Rightarrow $b = (\text{type of } b)(b+1);$

Arithmetic operators:-

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

→ If we are applying any Arithmetic 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 `S.o.println(10+0.0); // 10.0`

int + long = long `S.o.println('a'+'b'); 195`

long + float = float `S.o.println(100+'a'); 197`

double + char = double

char + char = int

Infinity:-

→ In the Case of Integral arithmetic (int, short, long, byte), There is no way to represent infinity. Hence, if the infinity is the result we will always get Arithmetic Exception. (AE = 1 by zero)

Eg:-

`S.o.println(10/0); R.E: AE: 1 by zero`

- But in Case of floating point arithmetic, there is always a way to represent infinity, for this float & Double classes Contains the following two Constants.

Positive-Infinity = infinity

Negative-Infinity = -infinity

$$\begin{array}{l} +ve\infty = \infty \\ -ve\infty = -\infty \end{array}$$

- Hence, in the Case of ~~Floating~~ floating point Arithmetic we won't get any Arithmetic Exception.

Eg:- ①. S.o.println(10/0.0) ; Infinity

②. S.o.println(-10/0.0) ; -infinity.

* NAN :- (Not a Number)

- In integral arithmetic. There is no way to represent undefined results. Hence, if the result is undefined we will get A.E in case of integral Arithmetic.

Eg:- S.o.p(0/0) ; RE: A.E: 1 by zero

- But in Case of floating point Arithmetic, there is a way to represent undefined results for this float & Double classes Contains NAN Constant.

- Hence, Even though the result is undefined we won't get any Runtime Exception in floating point Arithmetic.

Eg:- S.o.println(0/0.0); Nan.

* S.o.p(0.0/0); NaN

* S.o.p(-0/0.0); NaN

Ex:- public static void Sqrt (double d);

S.o.println (math. sqrt (4)); /2.0

S.o.println (math. sqrt (-4)); NaN.

→ For any x value including NaN the below Expressions always returns false, Except the (\neq) Expression returns True.

$$x \neq \text{NaN} \Rightarrow \text{True}$$

at $x=10$

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

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

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

S.o.p($\frac{float}{10} != \text{float.NaN}$); true.

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

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

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

Conclusion about A.E (Arithmetical Exception) :-

→ It is Runtime Exception but not Compiletime Error.

→ Possible only in Integral Arithmetic but not Floating point Arithmetic
(int, byte, short, char) (float, double)

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

3. String Concatenation Operator (+)

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

→ Sometimes it acts as arithmetic addition operator & Some time acts as String arithmetic Operator. (or) String Concatenation operator.

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

String d = "Shanth";

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

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

S.o.p(d+a+b+c); Shanth102030

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

$\frac{d+a+b+c}{Shanth10+20+30}$
 $Shanth1020+30$
 $Shanth102030$

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

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

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

String c = "Shanth";

$\times a = (b + c)$ ^{total String} C.E:- Incompatible type : found : String

Required : int

✓ $c = a + c$; ^{total String}

✓ $b = a + b$; ^{int}

$\times c = a + b$; C.E:- Incompatible type:

found : int

Required : String.

Relational Operators

$$A=65, a=97 \quad 41$$

These are $>$, $<$, \geq , \leq

→ We can apply Relational operators for Every primitive datatype.

Except boolean.

Eg:-

1) $10 > 20$ false ✓

2) $'a' < 'b'$ true ✓

3) $10 \geq 10.0$ true ✓

4) $'a' < 125$ true ✓

5) true \leq true ✓

6) true $<$ false ✓

CE:- Operator \leq can't be applied to boolean, booleans

→ We can't apply relational operators for the object types.

Eg:- 1) "Shanthi" $<$ "Shanthi" X

2) "durga" $<$ "durga123" X

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

→ Nesting of Relational operators we are not allowed to apply.

Eg:- ✓ S.o.p (10 $<$ 20);

✗ S.o.p (10 \leq 20 \leq 30)

boolean

CE:- Operator \leq can't be applied to boolean.

Eg:-

String s₁ = new String("durga");

String s₂ = new String("durga");

S.o.ptn(s₁ == s₂); false (reference)



S.o.ptn(s₁.equals(s₂)); true (content)



Equality Operators ($==$, $!=$)

→ These are $==$, $!=$

* We can apply Equality operators for Every primitive type including

boolean types.

Q.P

Ex:-	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 t_1 and t_2 if $t_1 == t_2$ returns True

iff both t_1 & t_2 are pointing to the same object.

i.e., Equality operator ($==$) is always meant for reference / address comparison

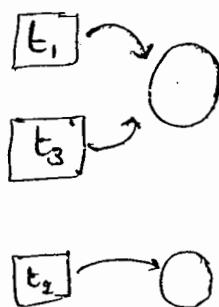
Ex): Thread $t_1 = \text{new Thread();}$

Thread $t_2 = \text{new Thread();}$

Thread $t_3 = t_1;$

✗ S.O.P($t_1 == t_2$) ; False

✓ S.O.P($t_1 == t_3$) ; True



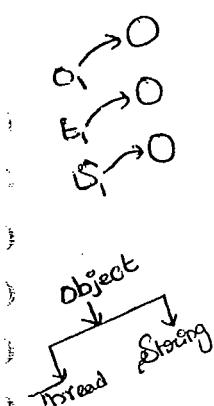
* To apply Equality Operators b/w the object references Compulsory

There should be some relationship b/w argument types.

[either parent to child (or) child to parent (or) Same type] otherwise

We will get CE: InComparable type], <http://javabynataraj.blogspot.com> 79 of 255.

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



Thread t₁ = new Thread();

String s₁ = new String("short");

s₁.op(t₁ == s₁); CE :- InComparable types Thread &

java.lang.String

s₁.op(t₁ == o₁); F

s₁.op(s₁ == o₁); F

→ For any object reference g₁, if g₁ is pointing to any object

g₁ == null is always false, otherwise g₁ contains null value

→ So, null == null is always True.

Note:-

* In General, == operator meant for Reference Composition

where as equals() method meant for Content Composition.

instanceof operator

(instanceof) ✓

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

SyD:-

g₁ instanceof X

any reference type

class / interface.

instanceof
HashMap
String

Ex:- short s=15;

Boolean b;

b = (s instanceof Short)

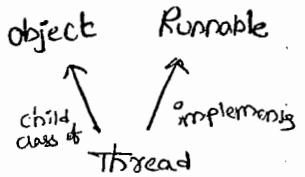
b = (s instanceof Number)

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, Compulsory there should be Some relationship b/w assignment 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 returning "false".

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

Eg:- Iterator iter = l.iterator();
while (iter.hasNext()) {

Object o = iter.next();

if (o instanceof Student)

{ Apply student related function }

else if (o instanceof Cu)

{ Apply customer related }

<http://javabynataraj.blogspot.com> 81 of 255.

Bit-Wise Operations :-

(sg) arguments

- (1) $\&$ → AND \rightarrow if Both operands are True then Result is True
- (2) $|$ → OR \rightarrow if atleast 1 operand is T " T
- (3) \wedge → X-OR \rightarrow if Both operands are different " T

e.g. S.o.println(4 & 5); 4

S.o.println(4 | 5); 5

S.o.println(4 ^ 5); 1

Ex:-

S.o.println(4 & 5); 4

$$\begin{array}{r} 100 \\ 101 \\ \hline 100 \end{array} = 4$$

S.o.println(4 | 5); 5

$$\begin{array}{r} 100 \\ 101 \\ \hline 101 \end{array} = 5$$

S.o.println(4 ^ 5); 1

$$\begin{array}{r} 100 \\ 101 \\ \hline 001 \end{array} = 1$$

→ We can apply these operations even for **integral data-types** also.

also.

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

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

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

Bitwise Complement Operator (\sim) :-

`S.oopl(nT);` CE: operator `n` can't be applied to boolean.

- ② We can apply Bitwise Complement Operator only for integral types, but not for boolean type.

Ex:- 1) S.o.println(\sim true);

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

✓ 2) S.o.pln(04); -5

$$4 \equiv 0000\ 0000 \quad \dots \quad 0100$$

$$24 = \boxed{1111\ 1111 \quad \dots \quad 1011}$$

$0 \rightarrow +ve$
 $1 \rightarrow -ve$

\downarrow 2's Complement
-ve

One's Comp

000 0000 - - - 0100

2's Comp

~~000~~ - - - - 0101

add 'i' to its comp

is g's Comp

-Ve 5

5

Note:

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

→ +ve no. will be represented directly in the memory. whereas as

-ve no's will be represented in 2's Complement form.

Boolean Complement Operator (!) :-

84

→ We can apply these operators only for Boolean type but not for integral types.

Ex:- (1) S.o.p(! u);

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.

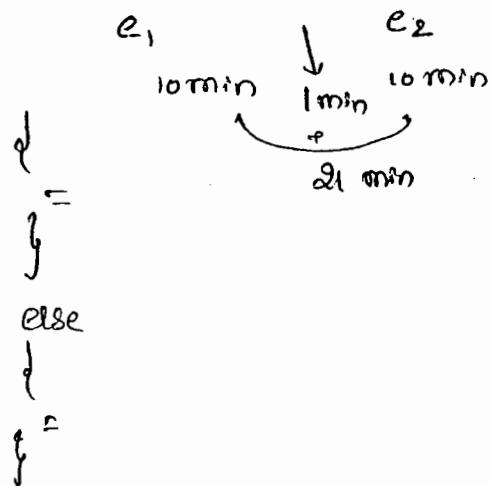
! ⇒ we can apply only for boolean types but not for integral types.

** 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. 2 nd 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 (num & num)



- 1) $x \& \& y \Rightarrow y$ will be evaluated iff x is True.
- 2) $x || y \Rightarrow y$ will be evaluated iff x is false.

Ex:-

```

int x=10;
int y=15;
if (++x>10 & ++y<15)
{
    ++x;
}
else
{
    ++y;
}
System.out.println(x+ "----+" +y);

```

Q.P.

	x	y
&	11	17
	12	16
	12	15
	11	17

```

② int x=10;
    if (x++ < 10) && (x/0 > 10)
    {
        S.o.println("Hello");
    }
    else
    {
        S.o.println("Hi");
    }

```

Ans:

- a) C.E
- b) R.E : Arithmetic Exception : 1 by Zero.
- c) Hello
- d) Hi

Note:

If we Replace $\&\amp;$ with $\&$

Then Result is ④, that is R.E.

$$\begin{array}{l} a=97 \\ A=65 \end{array}$$

TypeCast Operators:-

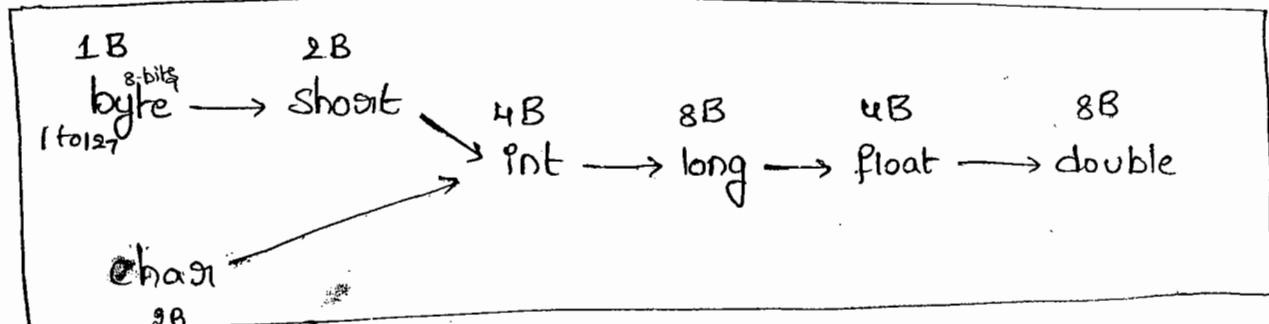
→ There are 2 types of primitive type Castings.

1. Implicit type Casting
2. Explicit type Casting.

Implicit Type Casting:-

- 1) Compiler is responsible to perform this type casting.
- 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 type casting



Ex(1):

① `double d=10;` [Compiler Converts int to double automatically]
 ↙ `S.o.println(d); 10.0`

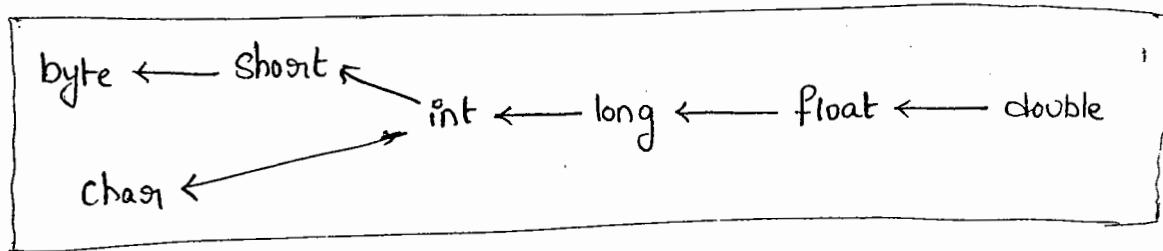
② `int x='a';` [Compiler Converts char to int automatically]
 ↙ `S.o.println(x); 97`

`A=91, B=98 ---`

`A = 65, B = 66, C = 67,`

2) Explicit type Casting :-

- 1) programmer is responsible to perform this TypeCasting
- 2) It is required whenever 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) $x \mid \text{byte } b = 130$
c-E: Possible loss of precision
found : int

Required : byte

2) $\text{byte } b = (\text{byte}) 130;$
S.o.p(b); -126

→ When ever we are assigning Bigger datatype value to the Smaller datatype variable then the most significant bit will be lost.

① X 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

47

$$130 \equiv 0000 \dots \underline{10000010}$$

(32-bit)

$$\text{byte } b \equiv \underline{10000010} \text{ (8 bit)}$$

↓ j's Complement
-ve

$$\begin{array}{r} 1111101 \\ \underline{-1} \\ 1111110 \end{array}$$

$$\begin{array}{l} 0000010 \\ \swarrow \\ 1111110 \end{array}$$

$$\begin{aligned} &= 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 \\ &= 64 + 32 + 16 + 8 + 4 + 2 + 0 \end{aligned}$$

∴ -126

$$\therefore -126$$

②

int i = 150;

short s = (short) i;

s.o.println(s);

$$150 \equiv 0000 \dots 010010110$$

32 bits

short s \equiv 0000 ... 00010110 \rightarrow 2 Bytes = short = 16-bits

↓ don't apply j's Comp.

+ve

$$\therefore s = 150$$

③ int x = 150;

byte b = (byte) x;

short s = (short) x;

s.o.println(b); -106

s.o.println(x); 150

$$150 \equiv 0000 \dots 010010110$$

$$\text{byte } b = \underline{00010110}$$

-ve

$\times 2^7 \text{ com}$

$$\underline{\underline{1101010}}$$

$$\begin{array}{r} 1101001 \\ \underline{-1} \\ 1101010 \end{array}$$

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

10/2/11

→ whenever we are assigning floating point datatype values to the integral datatypes 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.println(a); 130
```

```
S.o.println(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 C.E. & d)

Ex:- int b, c, d;
 a = b = c = d = 20 } ✓

3. Compound assignment operator :-

→ Sometimes we can mix assignment operator with some other operators to form Compound assignment operator.

Ex:- int a = 10;
 a += 30;
 System.out(a); 40

a += 30
 a = a + 30
 a = 10 + 30
 a = 40

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

+ =	& =	>> =
- =		>>> =
% =		<< =
* =		
/ =		

(10)

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

Ex(1)

~~byte b=10;~~
~~b=b+1;~~
S.o.println(b);

C.E.: PLP

found: int

Required: byte

b=b~~1~~;

✓ byte b=10;
b++;
S.o.println(b); //

byte b=10
b+=1;
S.o.println(b) ≠ 11

byte b=127; ✓

b+=3;

S.o.println(b); -126

Ex(2):

int a, b, c, d;

a=b=c=d=20;

a+=b*=c+=d+=2;

S.o.println(a + "----" + b + "----" + c + "----" + d);

620

600

30

10

Conditional Operator (?:)

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

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

int x = (a > b) ? 40 : 50;

F

S.o.println(x); 50

a+b → binary operator

++a → unary "

(a>b)? a:b; → ternary.

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 $\alpha = (\alpha > 50) ? 777 : ((b > 100) ? 888 : 999);$
 $\text{S.o.println}(\alpha); \quad 999$

Ex! -

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

<input checked="" type="checkbox"/> byte c = (true) ? 40 : 50;	$a < 12 \rightarrow T$ $a < b \times c \rightarrow E$
<input type="checkbox"/> byte c = (false) ? 40 : 50;	$a < b \times c \rightarrow E$ don't compare these variables

X | byte c = \downarrow (a < b) ? 40 : 50; C.E.: PLP
 | byte c = (a > b) ? 40 : 50; found: int

• **final int a=10, b=20;** required : byte .

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

New Operators :-

) → 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.

[] operators :-

→ We can use these operators for declaring & creating arrays.

Operator precedence :-

1. Unary operators:-

[] , $x++$, $x--$

$++x$, $--x$, ~ , !

new , < type > (used to type cast)

2. Arithmetic Operators:-

* , / , %

+ , -

3. Shift operators:-

>>> , >> , <<

4. Comparison operators:-

< , \leq , > , \geq , instanceof

5. Equality operators:-

== , !=

6. Bitwise operators:-

&

^

|

7. Short - Circuit operators:-

&&

||

8. Conditional operators:-

? :

9. Assignment operators:-

= , $+ =$, $- =$, $\cdot = \cdot \cdot \cdot$

Evaluation Order of operands :-

- There is no precedence for operands before applying any operators all operands will be evaluated from left to right.

Ex:-

class EvaluationOrderDemo

{

p.s.v.m (String [] args)

{

S.o.p (m,(1) + m,(2) * m,(3) + m,(4) * m,(5) / m,(6));

}

p.s.int m,(int i)

}

S.o.println(i);

return i;

}

}

o/p:-

10

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

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

$$1 + 6 + 20 / 6$$

$$1 + 6 + 3$$

$$7 + 3$$

$$= 10$$

Ex(2) :-

class Test

{

p.s.v.m (String[] args)

}

int x = 10;

x = ++x;

System.out.println(x);
}

1st increment

2nd place init into x

int x = 10;

x = x++;

System.out.println(x);
10

1st place x = 10

∴ x = 10++

∴ x = 11

but last operation is

x = 10

Ex(3) :-

① int x = 0;

(+2)³

x = $\frac{++x}{1} + \frac{x++}{2} + \frac{x++}{3} + \frac{++x}{4}$

System.out.println(x);
8

x = 0 1 2 3 4

x++ = 1

x++ = 2

3

4

Ex 4:-

int x = 0;

x += ++x + x++;

System.out.println(x);
2

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

= 0 + 1 + 1

x = 2

flow Control

16/05/2011
52

flow Control :-

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

flow - Control



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 :-

b) if - else :-

SyD :-

if (b)

Action if b is true

}

else

{

Action if b is false

→ The argument 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)  
{  
    System.out.println("Hello");  
}
```

else

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

C.E:- incompatible types

found : int

required : boolean

②

int x=10

```
if(x==20)  
{  
    System.out.println("Hello");  
}
```

else

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

③

int x=10;

```
if(x==20)
```

{

```
    System.out.println("Hello");  
}
```

}

else

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

}

O/P:- Hi

④

boolean b=false;

```
if(b==true)
```

{

```
    System.out.println("Hello");  
}
```

}

else

{

```
    System.out.println("Hi");  
}
```

}

O/P:- Hello

⑤

boolean b=false;

```
if(b==true)
```

{

```
    System.out.println("Hello");  
}
```

}

else

{

```
    System.out.println("Hi");  
}
```

}

✓

O/P:- Hi

- (2) Curly braces ({}) are optional and without curly braces we can take only one statement & which should not be declarative statement

Ex:-

if (true)

System.out.println("Hello");



if (true)

int x=0;



C.E!

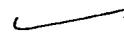
if (true)

int x=10;



✓

if (true);



Switch Statement :-

- If several options are possible then it is recommended to use if-else, we should go for Switch Statement.

Syn :- Switch (x)



Case 1 :

Action 1;

Case 2 :

Action 2;



default :

default Action;



- Curly braces are mandatory.

- both Case & default are optional inside a switch

Ex :-

int x=10;

Switch(x)



→ Within 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 argument are

byte

short

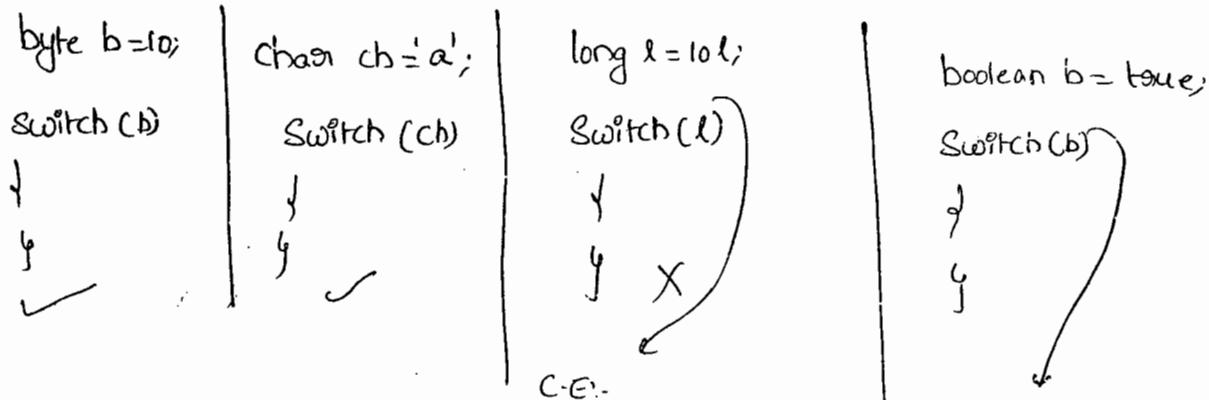
int

char

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

<u>1.4 v</u>	<u>1.5v</u>	<u>1.7v</u>
byte	⊕ Byte	
short	Short	⊕ String
char	Character	
int	Integer	
	+	
	enum	

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

Ex:-C-E:-**Possible loss of precision**

→ found: long

Required: int

C-E:-
In compatible types

→ found: boolean

Required: int

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

```

byte b=10;
switch (b)
{
    case 10:
        S.o.println("10");
    case 100:
        S.o.println("100");
    case 1000:
        S.o.println("1000");
}

```

-128 to
127

C-E:- possible loss of precision
→ found: byte int
Required: byte.

```

byte b=10;
switch (b+1)
{
    case 10:
        S.o.println("10");
    case 100:
        S.o.println("100");
    case 1000:
        S.o.println("1000");
}

```

→ Every Case label should be a valid Compiletime Constant, if we are taking a variable as Case label we will get Compiletime Error.

Ex:-

```
int x=10;
```

```
int y=20;
```

```
switch(x)
```

```
    |
```

```
Case 10:
```

```
s.o.println("10");
```

```
Case y:
```

```
}      |  
      | s.o.println("20"); X  
X      |
```

C.E! Constant Expression required.

Suppose final int y=20;

Case y:

```
s.o.println("20");
```

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

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

Ex:- int x=10;

```
switch(x+1)
```

```
    |
```

```
Case 10:
```

```
s.o.println("10");
```



```
Case 10+20:
```

```
s.o.println("10+20");
```

```
}
```

→ duplicate Case labels are not allowed.

e.g. int x=10;

Switch(x)



Case 97:

s.o.println("97");

Case 98:

s.o.println("98");

Case 99:

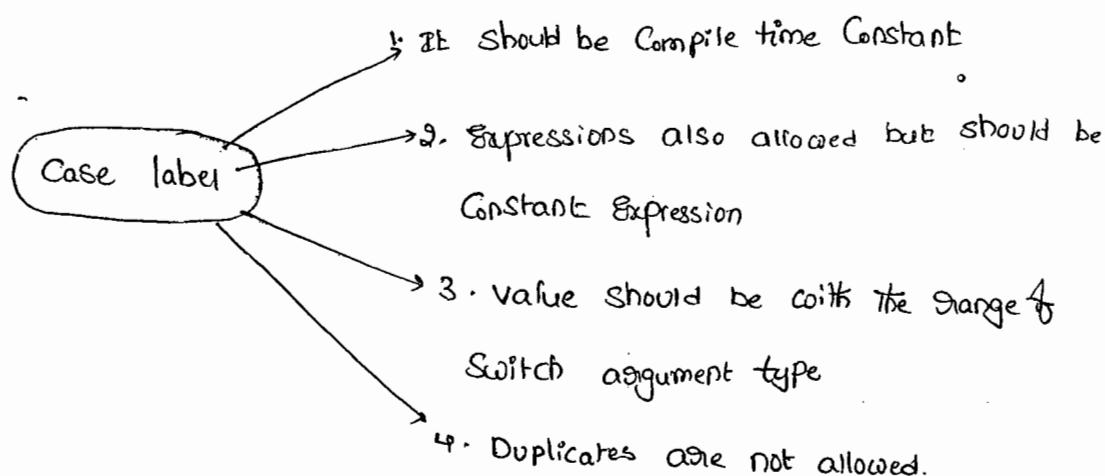
s.o.println("99");

Case 'a':

s.o.println("a"); X

C.E.: duplicate Case label

Summary :-



Fall-through inside switch :

→ Within 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.println ("0");`

Case 1:

`s.o.println ("1");`

`break;`

Case 2:

`s.o.println ("2");`

default:

`s.o.println ("def");`

}

Op!.

if $x=0$:-

0
1

if $x=1$:-

1

if $x=2$

2
def

if $x=3$

def

→ fall-through inside switch is useful to define some common action for several cases.

Ex:- `Switch(x)`

↓

Case 3:

Case 4:

Case 5:

`s.o.println("Summer");`

`break;`

Case 6:

Case 7:

Case 8:

Case 9:

`s.o.println("Rainy");`

`break;`

Case 10:

Case 11:

Case 12:

Case 13:

Case 14:

`s.o.println("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 anywhere within the switch but it is

Convention to take as Last case.

Ex:- `Switch(x)`

↓

default: `s.o.println("def");`

$$\frac{x=0}{0} \quad \frac{x=1}{2}$$

Case 0:

`s.o.println("0");`

`break;`

$$\frac{x=2}{2} \quad \frac{x=3}{def}$$

Case 1:

`s.o.println("1");`

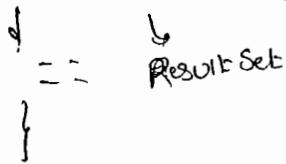
Case 2: `s.o.println("2");`

(b) Iterative Statements :-

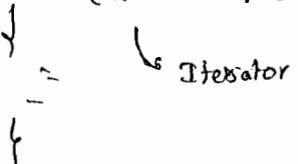
(i) while :-

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

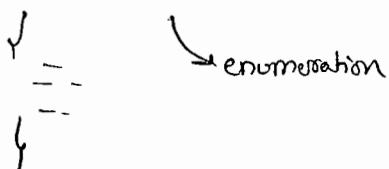
Ex:- ① `while (rs.next())`



② `while (itr.hasNext())`



③ `while (e.hasMoreElements())`



Syntax :-

`while (b)` → boolean type

↓

Action

{

→ The argument 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.println("Hello");`

{

C.E :- Incompatible types

→ found : int

required : boolean

→ C-style braces are optional and without C-style braces we can take only one statement which should not be declarative statement.

Ex(1)

while (true)

S.o.println("Hello");



while (true);



while (true)

int x=10;



while (true)

int x=10;



Ex(2) :

① while (true)



S.o.println("Hello");



S.o.println("Hi");



C.E :- unreachable statements

② while (false)



S.o.println("Hello");



S.o.println("Hi");



C.E :- unreachable statements

③

int a=10, b=20;

while(a < b)



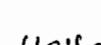
S.o.println("Hello");



S.o.println("Hi");



S.o.println("Hello");

O/P :- Hello
Hello
Hello

;



;

④ final int a=10, b=20;

while (a < b)



True



S.o.println("Hello");



S.o.println("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

→ Curly braces are optional & without having curly braces we can take only one statement b/w do & while which should not be declarative statement.

Ex:- ① do
 S.o.println("Hello");
 while(true);
 ✓

② do ;
 while(true);
 ✓

③ do
 int x=10;
 while(true);
 ✗

④ do
 {
 int x=10;
 }
 while(true);
 ✓

⑤ do
 while(true);
 X C.E!- → Compulsory one statement declare (or)
 take ;

⑥ do while(true)

```
S.o.println("Hello");  
while(false);
```

O/P:- Hello

Hello

,

/

Note! -

";" is a valid java statement

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

Ex-①

```

do      X
|
S.o.println("Hello");
|
}
while (true);
X S.o.println("Hi");
C.E!    Unreachable Statement
  
```

②

```

do      ✓
|
S.o.println("Hello");
|
}
while (false);
S.o.println("Hi");
O/P:-  Hello
        Hi
  
```

③

```

int a=10, b=20;
do
|
S.o.println("Hello");
|
}
while (a < b);
S.o.println("Hi");
O/P:-  Hello
        Hello
  
```

④

```

int a=10, b=20;
do
|
S.o.println("Hello");
|
}
while (a > b);
S.o.println("Hi");
O/P:-  Hello
        Hi
  
```

⑤ final int a=10, b=20;

```

do
|
S.o.println("Hello");
|
}
while (a < b);
X S.o.println("Hi");
C.E!:- unreachable
        statement
  
```

⑥

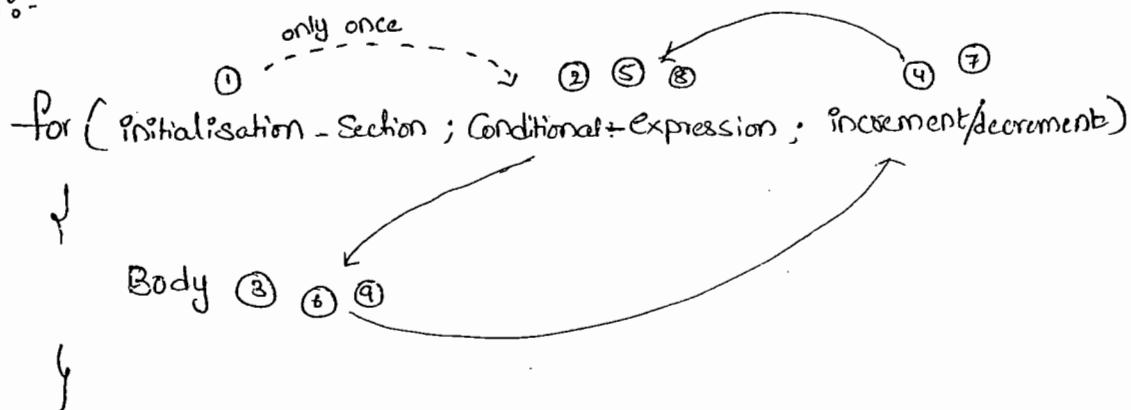
```

final int a=10, b=20;
do
|
S.o.println("Hello");
|
}
while (a > b);
S.o.println("Hi");
O/P:-  Hello
        Hi
  
```


for :-

→ This is the most commonly used loop

Syntax :-



- 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 just 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.println("Hello U R Sleeping"); i<3 ; i++)
{
    System.out.println(" No Boss U only sleeping");
}
```

O/P:- Hello U R 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".

Encaement & decarment Section :-

→ We can take any valid java Statement including S.O.P() also.

Ex:- int i=0;

```
for( System.out.println("Hello") ; i<3 ; System.out.println("Hi"))
{
    System.out.println(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:- $\text{for}(\text{; } \text{; }) ;$ Statement
So, it is True.

⇒ Represent infinite loop

Note:-

; is a valid Java Statement

Ex:-

$\text{for}(\text{int } i=0; \text{true}; i++)$	$\text{for}(\text{int } i=0; \text{false}; i++)$	$\text{for}(\text{int } i=0; ; i++)$
↓	↓	↓
$\text{s.o.println("Hello");}$	$\text{s.o.println("Hello");}$	$\text{s.o.println("Hello");}$
↓	↓	↓
$\text{s.o.println("Hi");}$	$\text{s.o.println("Hi");}$	$\text{s.o.println("Hi");}$
C.E:- unreachable	C.E:- unreachable	C.E:- unreachable
int a=10; b=20;	$\text{final int a=10; b=20;}$	
$\text{for}(\text{int } i=0; a < b; i++)$	$\text{for}(\text{int } i=0; a < b; i++)$	
↓	↓	True
$\text{s.o.println("Hello");}$	$\text{s.o.println("Hello");}$	
↓	↓	
$\text{s.o.println("Hi");}$	$\text{s.o.println("Hi");}$	X
O/P:- Hello ✓	O/P:- C.E! - unreachable statement.	
Hello		
!		

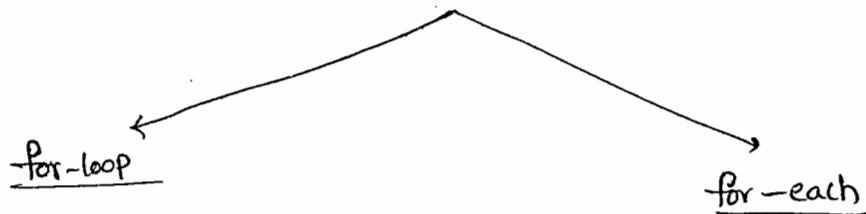
for-each() Loop :- (Enhanced for loop) :-

→ Introduced in 1.5v. This

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

Ex:- ① Point elements of Single dimensional Array by using General & enhanced for loops

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

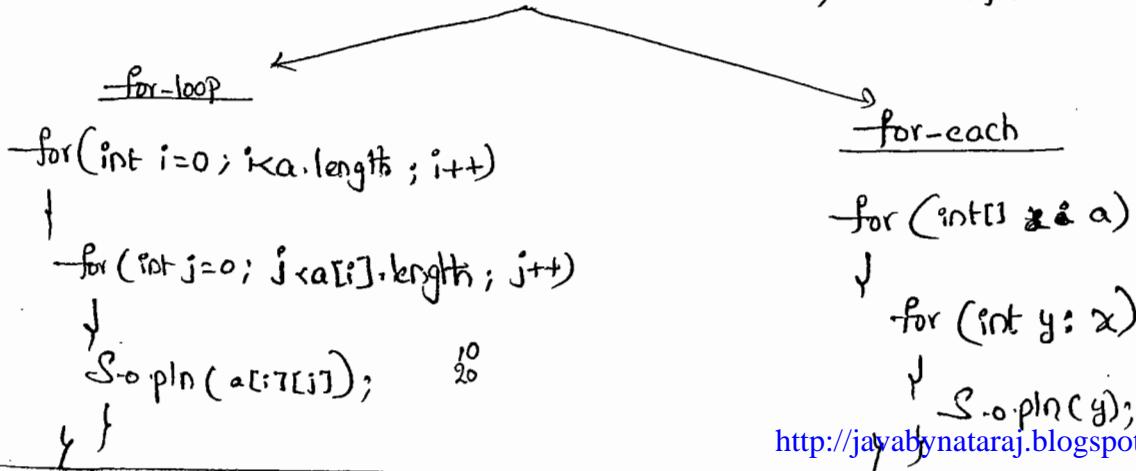


for(int i=0; i<a.length; i++)
{
 System.out.println(a[i]);
}
10
20
30
40
50

for(int x: a)
{
 System.out.println(x);
}
10
20
30
40
50

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

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



→ Even though for-each loop is more convenient to use, but it has the following **limitations**.

(i) It is not a general purpose loop -

(ii) It is applicable only for Arrays & Collections

(iii) By using for-each loop we should retrieve 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

(i) within the switch to stop fall through

(ii) inside loops to break the loop execution based on some condition

(iii) inside **labeled** blocks to break that block execution based on some condition.

Ex:-

Switch (b)

```

    ↓
    :
    break;
    :
  }
```

```

for (int i=0 ; i<10 ; i++)
  ↓
  if (i == 5)
    ↓
    break;
    System.out(i);
    :
  }
```

Class Test

}

P. S. V. M (→)

↓

int i=10;

L:

↓

System.out("Hello");

if (i == 10)

break L;

System.out("Hello")

else

System.out("End");

→ If we are using break Statement Any where else we will get
Compiletime Error

Ex:- Class Test

```
{  
    P-S-V-mc ( → )  
    |  
    int x=10;  
  
    if(x==10)  
        break; X  
        System.out.println("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;  
    System.out.println(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; → X
 S.o.println("Hello");
C.E:- Continue outside of loop

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:-

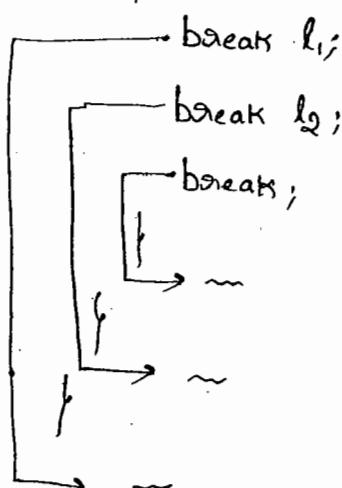
```

l1:
for(----)
↓
l2:
for(----)
↓
for(----)
↓
  
```

Ex 2:-

```

l1:
for(int i=0; i<3; i++)
↓
for(int j=0; j<3; j++)
↓
if (i==j)
break;
S.o.println(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)

x = 1

Ex:- int x = 0;

do
|

x++;

System.out.println(x);

if (++x < 5)

Continue;

x++;

System.out.println(x);

} while (++x < 10);

x = 0

1 x 5

x = 5
2 x 5

3 x 5

4 x 5

5 x 5

6 x 5

7 x 5

8 x 5

9 x 5

10 x 5

1
4
6
8
10

Imp Note!

→ Compiler will check for unreachable statements only in the case of loops but not in 'if - else'.

Ex:- ① if (true)
|
System.out.println("Hello");
{
else
{
System.out.println("Hi");
}

O/P:- Hello

② while (true)
|
System.out.println("Hello");
{
System.out.println("Hi");
}
o/p:- C.E.

Unreachable Statement

Declarations & Access Modifiers

① Java Source file Structure (1 - 9)

Package :-

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

|

|

Class C

|

|

Save: Sai.java ✓

R.java ✓

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 ✓

Case 2:-

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.println("A class main method");

}

Class B

{

P.S.v.m(String[] args)

{

S.o.println("B class main method");

}

Class C

↓

P.S.V.m(String[] args)

↓

S.o.println("C class main method");

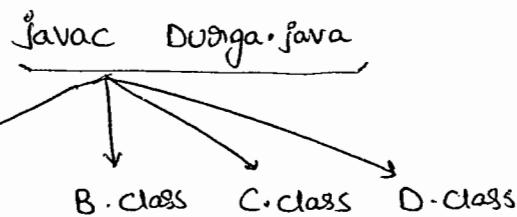
↓

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!

→ 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();
    }
}
```

→ Whenever you are using import statement it is not required to use fully qualified name 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

Import Statements

Explicit class import :-

Ex:- `import java.util.ArrayList;`

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

→ Best suitable for Hitch City where readability is important

Implicit class import :-

Ex:- `import java.util.*;`

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

→ Best suitable for Amrapet where typing 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 whether we are using those header files or not. Hence this is static loading.
- But in the case of Java language import statement no .class file will be loaded at the time of import statement, in the next lines of code whenever 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?

- X ① import java.util;
- X ② import java.util.ArrayList,*;
- ✓ ③ import java.util.*;"
- ✓ ④ import java.util.ArrayList;

Case 4:

→ Consider the Code,

```
class MyRemoteObject extends java.rmi.Unicast
    RemoteObject
    {
    }
```

→ The Code Compiles fine Even though we are not using import statement because we used fully Qualified 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.

Case 2:-

Example:-

```

import java.util.*;
import java.sql.*;

class Test
{
    public static void main(String[] args)
    {
        Date d = new Date();
    }
}

```



C-E:- Reference to Date is ambiguous.

Note:-

even in List Case also we will get the same ambiguity problem.

because it is available in both Util & Sql packages.

Case 3:-

```

import java.util.Date;
import java.sql.*;
class Test
{
    public static void main(String[] args)
    {
        Date d = new Date();
    }
}

```

order :-

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

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

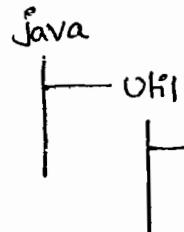
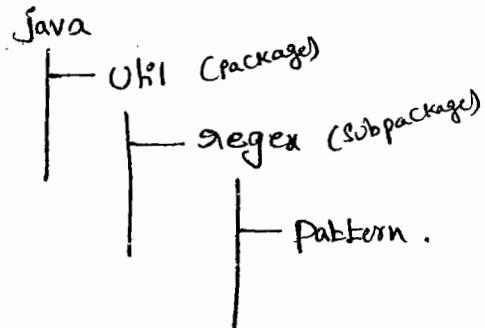
→ order See above

Date } available in both Util
List }

Case 7:-

→ When even 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 compilation 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:-

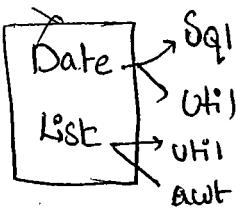
Without Static import

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



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

class Test

↓

p. static String name = "xyz";

}

Test.name.length();

↓ It is a method present in String class

Static variable

Present in Test class

of the type String

It is a class name

Name

Class System

↓

Static PrintStream OUT;

}

System.out.println();

↓ It is a method present in PrintStream class.

It is a static variable

of the type PrintStream

present in System class

It is a class

Present in

java.lang package

Explanation:-

→ OUT is a Static variable present in System class hence we can access by using className.

→ But whenever 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) :-

Ex:- `import static java.lang.Integer.*;`

`import static java.lang.Byte.*;`

Class Test

{

`p.s.v.m(String[] args)`

{

`s.o.println(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 Resolving 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.println(MAX-VALUE);

}

}

→ If we are Commenting Line ① Then Explicit Static import will get Priority Hence we will get Integer class MAX-VALUE is o/p 127.

→ If we are Commenting Lines ① & ③ Then Byte Class MAX-VALUE will be Considered & we will get 127 as o/p.

(-ve point) :-

→ Strictly Speaking usage of Class Name to access static variables & methods improves readability of the code. Hence it is not recommended to use Static import.

Q) Which of the following import statements are valid.

- X ① `import java.lang.math.*;` (we should not use * after the class).
- X ② `import java.lang.math.Sqrt.*;` (we should not use * after the method).
- X ③ `import static java.lang.math;`
- ✓ ④ `import java.lang.maths;`
- ✓ ⑤ `import static java.lang.maths.*;`
- X ⑥ `import static java.lang.math.Sqrt();` → problems
- ✓ ⑦ `import static java.lang.maths.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 members we can access directly.

Packages

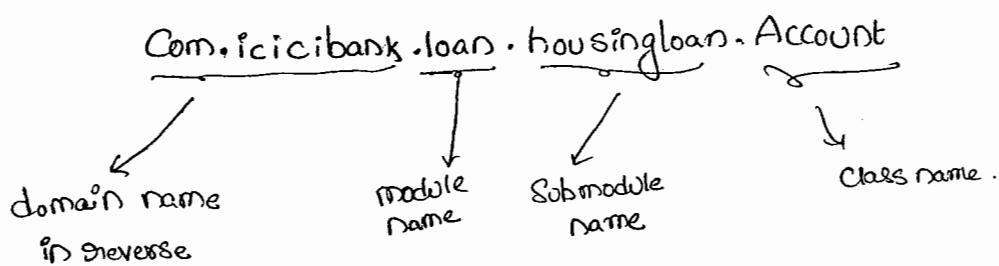
Java classes are
there in Java
according to 1.6v

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 outside persons can't access directly
- ③ It improves modularity of the application.

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



Ex:-

```
package com.duagajobs.itjobs;
```

```
public class HydJobs
```

```
{
```

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

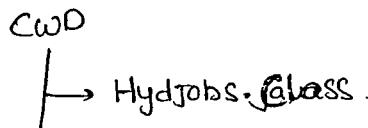
```
{
```

```
        System.out.println("Getting jobs is very easy");
```

```
}
```

① `javac HydJobs.java`

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

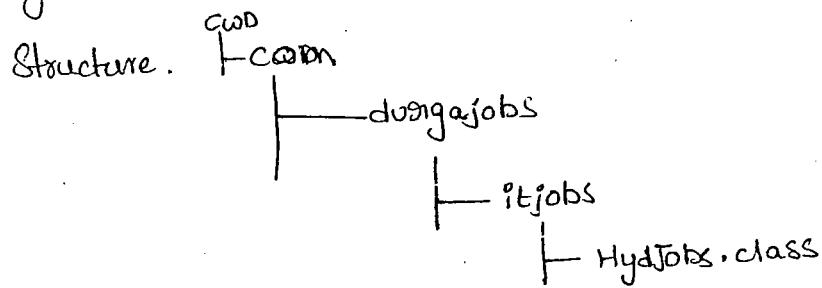


② `javac -d . HydJobs.java`

→ Destination
→ to place generated
class files

current
working
directory

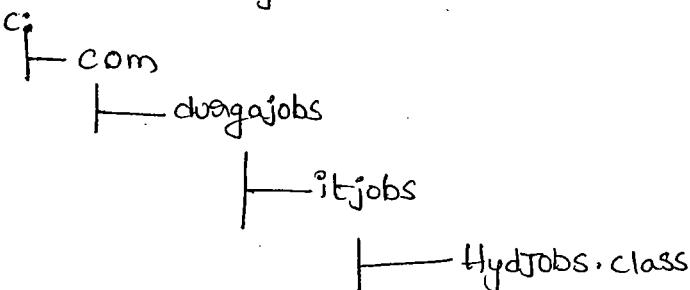
→ generated class file will be placed into Corresponding package



→ 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

Ex. `javac -d c: HydJobs.java`



→ 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 Compile time Error.

Run

→ Java com.dvorgasobs.itjobs.HydJobs ←

Ques:- Getting job is very easy.

Conclusions:-

① In Any Java program there should be only At most 1 package statement. If we are taking more than one package statement we will get **Compiletime Error**.

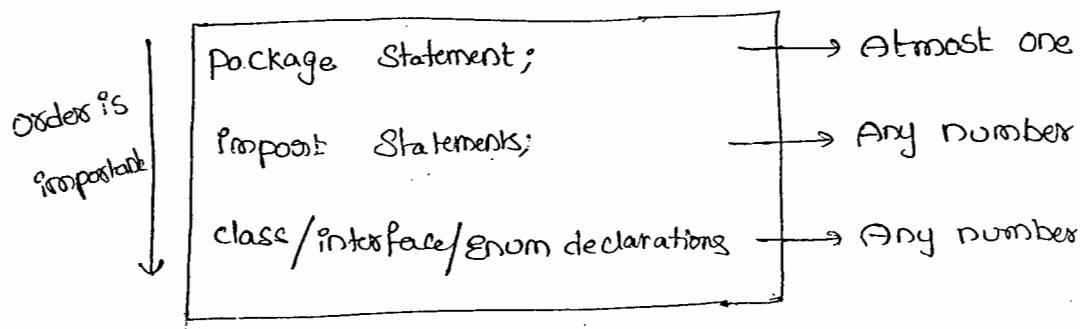
Ex:- ✓ package pack1;
 → package pack1; ←
 Class A
 {
 }
 C.E:- class, interface or enum expected.

② In Any Java program the First non Comment Statement Should be package statement (if it is available).

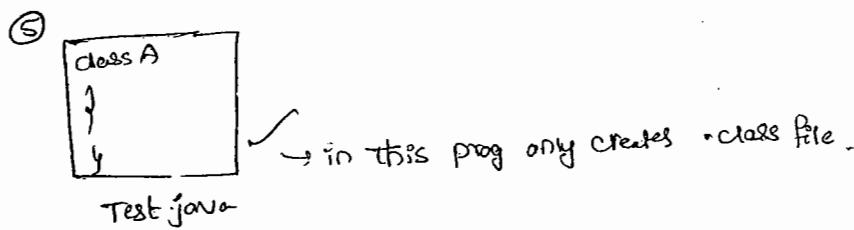
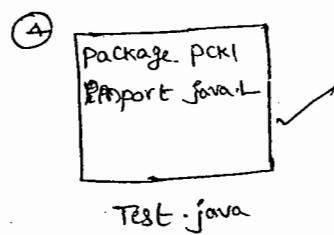
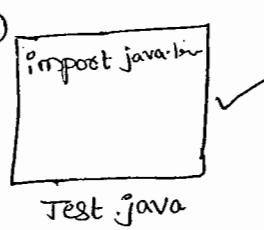
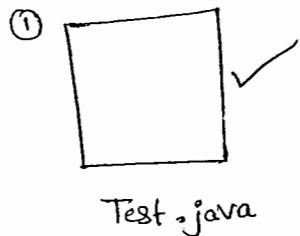
Ex:- ✓ import java.util.*;
 → package pack1;
 Class A
 {
 }
 C.E:- class, interface or enum Expected.

→ The proper structure of a Java Source file is

78



→ The following are Valid Java programs.



→ An Empty Source file is a Valid Java program.

**** Class modifiers ****

→ whenever we are creating our own java class compulsory we have to provide some information about our class to the JVM

Like,

- Can be
- (1) whether our class accessible from anywhere or not.
 - (2) whether child class creation is possible for our class or not.
 - (3) whether instantiation 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 xxxxxxxx not allowed here".

Ex:- private class Test

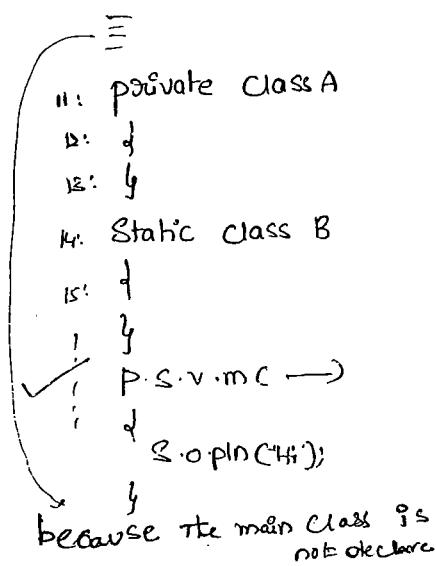
```
p.s.v.m(____)
{
    int x=0;
    for(int y=0; y<3; y++)
    {
        x=x+y;
        System.out.println(x);
    }
}
```

C.E! - modifier private not allowed
here

七三

→ 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.



Access Specifiers Vs access modifiers :-

28/04/11

- 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 anywhere.

Ex.

Package pack1;

Public class A

1

Public void mi()

1

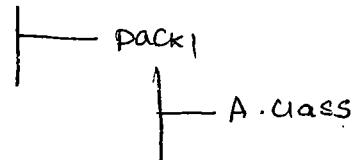
```
g.o.println("Hello");
```

二

```
s.o.pin('Hello');
```

javac -d . A.java

CDWID



```

package pack2;
import pack1.A;
class B
{
    public static void main(String[] args)
    {
        A a = new A();
        a.m1();
    }
}

```

Comp. 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 Compiling B class, Saying "pack1.A is not public in pack1; Can't be accessed from outside Package"

default classes:-

→ If a class declared as default then we can access that class only within 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 override that method in the child class.

Ex:-

```

Class P
|
public void property()
|
{
    System.out.println("money + Gold + Land");
}

public final void maaay()
{
    System.out.println("Subba laxmi");
}

```

C.E

Class C extends P

```

public void maaay()
{
    System.out.println("Kajal | Zsiba | Atara");
}

```

C.E! - maaay() in C Cannot override maaay() 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

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 by default.

but Every variable present in final class need not be final.

→ The main Advantage of final keyword is we can achieve 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 :-

→ Even though we don't 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 Compulsory ends with ; .

Ex:-

- 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 getNoOfWheels()
    {
        return 6;
    }
```

```
Class Auto extends Vehicle
```

```
    public int getNoOfWheels()
    {
        return 3;
    }
```

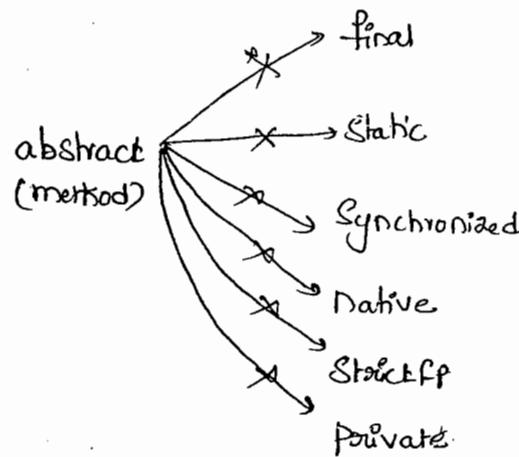
→ By declaring abstract methods in parent class we can define

Guidelines to the child classes which describes the methods those

are to be Compulsory 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 instantiation Then we have to declare that class as abstract. i.e., for abstract classes instantiation (creation of object) is not possible.

Ex:- abstract class Test

↓

↓

Test t = new Test();

C:E:- Test is abstract; Cannot be instantiated

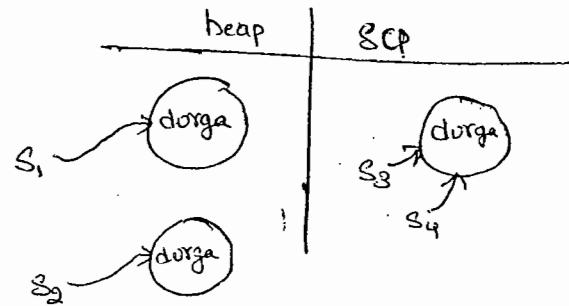
Test t = new Test();

Ex: String s₁ = new String("durga");

String s₂ = new String("durga");

String s₃ = "durga";

String s₄ = "durga";



- 1) notify() & notifyAll()
- 2) Collection & Collections
- 3) equals() & ==
- 4) Comparable & Comparator
- 5) String & StringBuffer
- 6) StringTokenizer & StringTokenizer
- 7) Throw & Throws
- 8) throws & thrown
- 9) HashMap & Hashtable
- 10) enum, Enum, Enumeration
- 11) final, finally, finalizer

✓ 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 (r4)

17) I18N

18) enum

19) development

Dell

chaitanyaacharya@gmail.com

Satish4dworld@ " "

29444524

Chait

Chaitanya-Anumanchi@dell.com.

ON purnagajobsInfo 9870807070

5

abstract class Vs abstract method :-

7.7

- If a class Contains atleast one abstract method then Compulsory That class should be declared as abstract otherwise we will get **Compile-time Error**. because, The implementation is not Complete & hence We Can't Create an object.
- Eventhough This class doesnot Contain any abstract method still we can declare the class as abstract. i.e, abstract class Can Contain Zero or no. of abstract method.

Ex:- HttpServlet, This class doesn't Contain any abstract method but still it is declared as abstract.

Ex:-

① Class Test

↓

public void m();

↳ C.E:- missing method body, or declare abstract

② Class Test

↓

public abstract void m();

↓

↳ C.E :- abstract methods Can't have a body

③ Class Test

↓

public abstract void m();

↓

C.E :- Test is not abstract and doesn't have abstract method m() in Test.

Ex-4:-

Abstract class Test

```
}  
public abstract void m1();  
public abstract void m2();  
{  
class SubTest extends Test  
{  
public abstract void m1(); }  
}
```

C.P.:- SubTest is not abstract and does not override abstract method m1() in Test

→ We can handle these Comptime Errors either by declaring SubTest as abstract or by providing implementation for m1().

Note:-

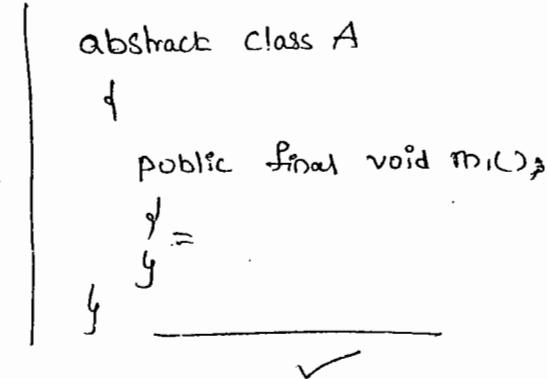
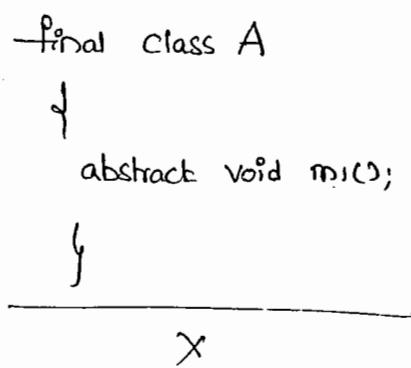
→ The usage of abstract methods, abstract class & interfaces are Recommended & it is always good programming practice.

Abstract Vs Final :-

→ Abstract methods we have to override in child classes to provide implementation. whereas 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 whereas abstract class can contain final methods.



strictfp (all lower case) modifiers :- (strictfloatingpoint)

- strictfp is the modifier applicable for methods & classes but not for variables.
- if a method declared as strictfp all floatingpoint calculations in that method has to follow IEEE 754 Standard So, that we will get platform independent results.
- strictfp, ^{method} always talks about implementation whereas abstract method never talks about implementation. Hence strictfp-abstract method combination is illegal combination for methods.
- If a class declared as strictfp then every concrete 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 struct<fp void m(); 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 m()  
    {  
        System.out.println("Hi");  
    }  
}
```

```
Package pack2;  
import pack1.A;  
class B  
{  
    A a = new A();  
    a.m();  
}
```

→ Even though m() method is public, we can't access m() 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 within the current class.
- abstract methods should be visible in child classes to provide implementation whereas 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 within the current package anywhere but outside package only in child classes.

Protected = <default> + kids of another package
(only child reference).

- * within 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

Ex:-

```
package pack1;
```

```
public class A
```

```
↓
```

```
protected void m1()
```

```
↓
```

S.O..println("The most misunderstood modifier in Java");

```
↓
```

Class B extends A

```
↓
```

P.S.V.m(→)

```
↓
```

✓ A a = new A();

✓ | a.m();

✓ B b = new B();

✓ | b.m();

✓ A a; = new B();

✓ | a.m();

→ The most restricted modifier
is "private"

→ The most accessible modifier
is "public"

→ private < default < protected
< public

→ The recommended modifier for
variables is private

Yellow circle: The recommended modifier for
methods is public

Package pack2;

Import pack1.A;

public class C extends A

↓

P-S-V.m(—)

↓

A a = new A();

X a.m();

C c = new C();

✓ c.m();

A a; = new C();

X a.m();

↓

FACT 1

A | ~~default~~

|
| protected void m();

package2

| B extends A

|
| C extends B

| ✓

package3

| D extends B

| ✓

→ The most restricted

* **private < default < protected < public**

Visibility	private	<default>	protected	public
① within 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 whether 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 been initialized.

have

Rule:-

For the final instance variables we should perform initialization before constructor completion.

→ 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 anywhere 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 should perform initialization explicitly otherwise we will get C.E.

Ex:-

Class Test



static int x;



Class Test



final static int x;



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

Rule:-

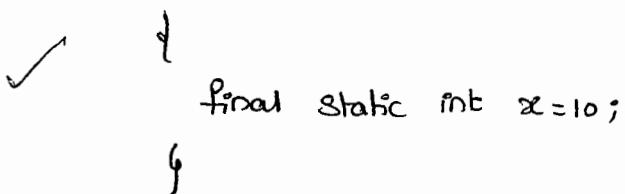
- * For the final static variables we should perform initialization

Before Class loading Completion.

- * i.e., the following are various places to perform this

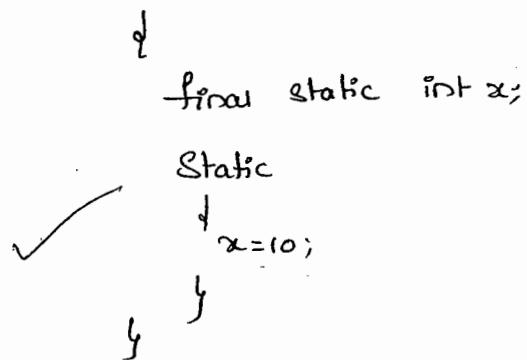
① At the time of declaration

e.g:- Class Test



② Inside Static Block

e.g:- Class Test



→ If we are performing initialization anywhere else we will get Compiletime Error.

Class Test

 ↴
final static int x;

 public void m1()

 ↴
 x=10; X

 ↴
C.E! - Can't assign a ^{value} variable to final
variable x.

iii) Final Local Variables :- Ques

→ for the local variables JVM won't provide any default values.

Compulsory we should perform initialization before using that variable.

Ex:- Class Test

```
① public void main() {  
    int x;  
    System.out.println("Hello");  
}
```

Op! - Hello

② Class Test

```
public void main() {  
    int x;  
    System.out.println(x);  
}
```

C.E! - variable x might not have been initialized.

→ Even though Local variable declared as the final it is not required to perform initialization if we are not using that variable.

Ex! Class Test

```
public static void main() {  
    final int x;  
    System.out.println("Hello Sun");  
}
```

Op! - Hello Sun.

* The only applicable modifier for local variables is final. If we are using any other modifier we will get Compiletime Error.

Ex:- Class Test

```
public static void main() {  
    public int x=10;  
    private int x=20;  
}
```

Static int x=60; X
Protected int x=30; X
<http://javabynataraj.blogspot.com> 160 of 255.
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.mC →



m,(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.pn(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 separate 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.

The begining
first static variable is created at
when class is created.

Ex:- Class Test

{

int x=10;

Static int y=20;

P.S.V.M(—)

}

Test t₁ = new Test();

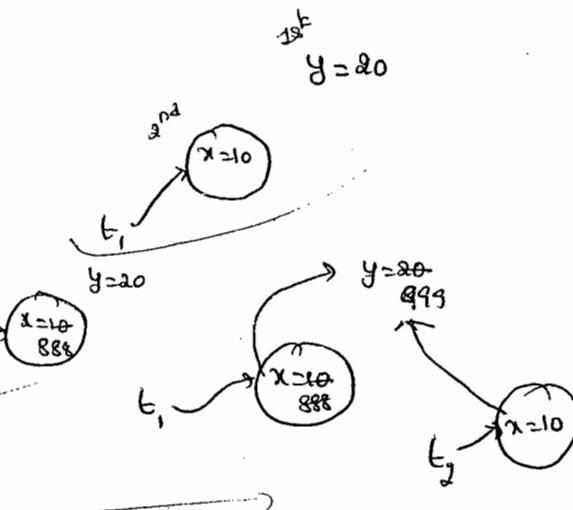
t₁.x = 888; →

t₁.y = 999; →

Test t₂ = new Test();

S.O.println(t₂.x + " --- " + t₂.y);

} 10



for every object a
separate copy will be
created.

↳

- Static members can be accessed from both instance & static areas whereas 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.println(x);

y

IV. public static void m1()

↓
S.O.println(x);

y

→ which of the above we can take simultaneously within the same class.

✓ A) I & III

✗ B) I & IV. CE! - Non-Static variable x can not be accessed from static context

✓ C) II & III

✓ D) II & IV

✗ E) I & II

✗ F) III & IV

→ For static methods compulsory implementation should be available whereas for abstract methods implementation should not be available Hence abstract-static combination is illegal for methods.

→ → for static methods overloading concept is applicable Hence within the same class we can declare 2 main methods with different arguments

Ex:- Class Test

↓

p. S. v.m (String[] args)

↓

S.o.println("String[]");

↓

public static void main (int[] args)

↓

S.o.println("int[]");

↓ } {

O/P:- String[]

→ But JVM also

→ But Jvm always calls static assignments 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 does not contain main method then the parent class main method will be executed.

Ex:- Class P



P.S.V.m (String[] args)



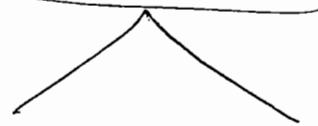
S.o.println ("Parent Class");



Class C extends P



javac p.java



P.class

%p java p

Parent class

%p java C

parent class.

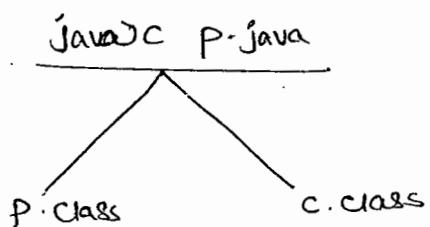
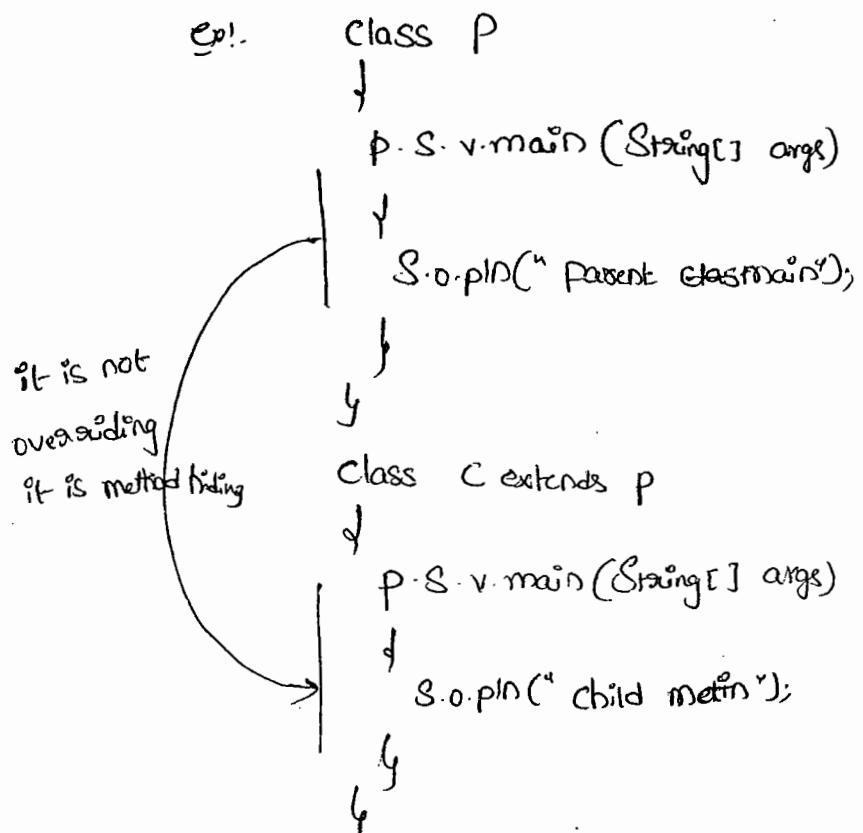
→ It seems that overriding Concept is applicable for static methods but it is not overriding, it is **method hiding**.

Ex:-

Class P



P.S.V.m (→)



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-
B

Class Native

۹

Static

2

⑤ Load

Native Library

3

System.loadLibrary("native Library")

191 Declares

public native void m();

a native
method

Class Shift

P·S·V·M(—)

3

⑤ Invoke a Native API = new Native();

Native method n.m.();

→ For native methods implementation is already available in other languages and we are not responsible to provide implementation. Hence native method declaration should Compulsory Ends with ";"

e.g. 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 whereas 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 guarantee 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 breaks platform independent nature of Java. because we are depending on result of platform dependent languages.

④ "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 inconsistency 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.

⑤ "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.

⑥ "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.

③ If a variable declared as volatile then for every thread a separate local copy will be created.

→ Every intermediate modification performed by that thread will take place in local copy instead of master copy.

→ Once the value gets 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 separate copy for every thread, increases complexity of the programming & 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 its value keep on changes whereas as 'final' variable means its value never changes. Hence final-volatile combination is illegal combination for variables.

Conclusion:

→ The only applicable modifier for local variables is final.

→ The modifiers which are applicable 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	Outer	Inner	methods	variables	blocks	interfaces	enum	Constructors
classes								
Public	✓	✓	✓	✓	X	✓	✓	✓
<default>	✓	✓	✓	✓	X	✓	✓	✓
private	X	X	✓	✓	X	✓	X	X
protected	✓	✓	✓	✓	X	✓	X	X
final	✓	✓	✓	✓	X	✓	✓	X
abstract	✓	✓	✓	✓	X	✓	✓	X
Static	✗	✗	✗	✗	X	✗	X	X
Synchronized	X	X	X	X	✓	✓	✓	✓
Native	X	X	X	X	✓	✓	X	X
Struct	X	X	X	X	✓	✓	X	X
transient	X	X	X	X	✓	✓	X	X
Volatile	X	X	X	X	✓	✓	X	X

→ 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
 - (i) method Naming Conflicts
 - (ii) Variable " "
- (6) Marker Interface
- (7) Adapter class
- (8) Abstract Class Vs Concrete Class vs Interface.
- (9) diff. b/w abstract class & interface

Interface:-

② Any Service requirement Specification (SRS) is Considered as Interface.

→ from the client point of view an interface defines the Set of Services what is expecting.

→ from the Service provider point of view an interface defines the Set of Services what is offering.

③ Hence an Interface Considered as Contract b/w Client & Service provider.

Ex:-

→ By using Bank ATM GUI Screen, Bank people will highlight 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 <http://javabynataraj.blogspot.com> 172 of 255.

- (i) we can achieve Security, because we are not highlighting our internal implementation.
- (ii) Enhancement will become very easy, because without affecting 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 Compulsory we should provide implementation for every method of that interface otherwise we have to declare class as abstract. Violation leads to Compile-time Error.

→ whenever we are implementing an interface method Compulsory it should be declared as public otherwise we will get CompiletimeError.

Extend 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 ^{or} and 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

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.

Q:

- 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

Ex:- 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 by default public & abstract the following modified are not applicable for interface methods.

(1) private

(2) protected

(3) <default>

(4) final

(5) static

(6) static fp

(7) synchronized

(8) native

→ Which of the following method declaration are valid inside interface?

- (1) public void m(); ✓
- (2) public static void m(); ✗
- (3) public synchronized void m(); ✗
- (4) private abstract void m(); ✗
- (5) public abstract void m(); ✓

Interface Variables:-

→ An interface can contain variables. The main purpose of these variables is to specify.

Constants 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 public int x=10;
- 7) public final int x=10;

g) Static final int $x=10$;

→ As interface variables are public static & final we can't declare with the following modifiers.

- (1) private (3) <default> (5) volatile .
- (2) protected (4) transient

→ for the interface variable compulsorily one 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) def 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;
```

```
    X
```

```
s.o.println(x);
```

```
}
```

```
C.E.:
```

```
Class Test implements Interf
```

```
{
```

```
p.s.v.m (String[] args)
```

```
{
```

```
int x=88;
```

```
}
```

```
s.o.println(x); 88
```

```
    ✓
```

Interface Naming Conflicts

1) Method naming Conflicts :-

Case 1 :-

→ 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()



Case 2 :-

→ 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.

Eg:-

interface Left



Public void m1();



interface Right



Public void m1(int i);



Class Test implements Left, Right



Public void m1()



Public void m1(int i)



Overloaded
methods

Case 3 :-

→ If Two Interfaces Contains a method with Same Signature but different return types. Then it is impossible to implement both interfaces at a time.

Ex:-

interface Left	interface Right
↓	↓
public <u>void</u> m1();	public <u>int</u> m1();
}	}

→ We Can't write any Java class which implements both interfaces simultaneously.

Q) Is it possible A Java class can implement any no. of interfaces simultaneously.

- * Yes, Except if Two interfaces Contains a method with same signature but different return types.

Q) Variable naming Conflicts :-

interface Left

↓

int x=888;

↓

interface Right

↓

int x=999;

↓

Class Test implements Left, Right

{

P.S.v.m ()

{

S.o.println(x);

}

}

C.E. - reference to x is ambiguous.

→ There may be a chance of 2 interfaces Contains available

with same name & may arise 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 other objects will get ability such type of

interfaces are called marker interface (or) Tag interface (or)

ability interface.

Ex:- Serializable, Clonable, RandomAccess, SingleThreadMode.

→ 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 won't 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 programming.
- Q)** Is it possible to create our own marker interface?
- A)** Yes, But customization of JVM is required.

Ex:- Sleepable, Eatble, 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 Adapter X implements X
{
    m1() {}
    m2() {}
    !
    m1000() {}
}
```

If we create an object for this empty result so for this class declare as abstract by default abstract.

→ If we implement an interface directly ~~as~~ Compulsory we should provide implementation for every method of that interface, whether we are interested 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 required method but not all this approach reduce length of the code & improves readability.

⇒ Class Test extends Adapter X

```

{
    m4()
    ...
}
```

Concrete 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

Ex:- **Servlet.**

→ we are talking about implementation but not completely (Just partially implementation) Then we should go for abstract class.

Ex:- Generic-Servlet

Http-Servlet

→ we are talking about implementation Completely & 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 requirement specification. Then we should go for interface.	1) If we are talking about implementation but not completely (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 modifiers i.e., we can use any modifier.

4) Every variable present inside interface is public, static final, by default whether we are declare or not.

5) For the interface variables we can't declare the following modifiers private, protected, transient, volatile.

6) For the interface variables Compulsory we should perform initialization at the time of declaration.
Only

7) Inside interface we can't take instance & static blocks.

8) Inside interface we can't take constructor.

4) abstract class variables need not be public, final, static.

5) There are no restriction for abstract class variable modifiers.

6) For the abstract class variables there is no restriction like performing initialization at the time of declaration.

7) Inside abstract class we can take static block & instance blocks.

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 whereas 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 programming practice we are miss using the role of abstract class.
→ we should bring abstract class into the picture whenever we are talking about implementation.

28/4/11

OOPS Concept

= = = = =

- 1) Data hiding 2
 - 2) Abstraction 2
 - 3) Encapsulation 2
 - 4) Tightly Encapsulated class 3
 - 5) IS-A Relationship 3
 - 6) Has-A Relationship 5
 - 7) method Signature 6
 - * 8) Overloading 7
 - 9) Overriding 10
 - 10) Method hiding 14
 - 11) Static Control flow 18
 - 12) Instance Control flow 22
 - 13) Constructors 24
 - 14) Coupling 42
 - 15) Cohesion 43
 - 16) Type-Casting -40
- polymorphism - 17
- Type-Casting = 40

① Data Hiding :-

- Hiding of the data, So that outside person can't 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 achieve **Security**.

② Abstraction :-

- Hiding internal implementation details & just highlight the set of services what we are offering, is called "Abstraction".

Ex:-

- By Bank ATM machine, Bank people will highlight the set of services what they are offering without highlighting internal implementation. This concept is nothing but Abstraction.

- By using interfaces & abstract classes we can achieve abstraction.

- The main **Advantages of Abstraction** are.

- 1) We can achieve Security as no one is allowed to know our internal implementation.

- 2) Without affecting 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.

→ whether the class contains getter & setter methods are not & whether those methods declared as public or not these are not required to check.

Ex:-

```
Class A
{
    private int balance;
    public int getBalance()
    {
        return balance;
    }
}
```

Q:- Which of the following classes are Tightly Encapsulated.

```
✓ Class A
  |
  | private int x=10;
  |
  ✓ Class B extends A
  |
  | int y=20;
  |
  ✓ Class C extends A
  |
  | private int z=30;
```

→ 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 follows Data Hiding & Abstraction such type of class is said to Encapsulated class.

Encapsulation = Data Hiding + Abstraction

Ex:-

Class Account

{

 private double balance;

 public double getBalance()

 {

 // validate user

 return balance;

 }

 public void setBalance(double balance)

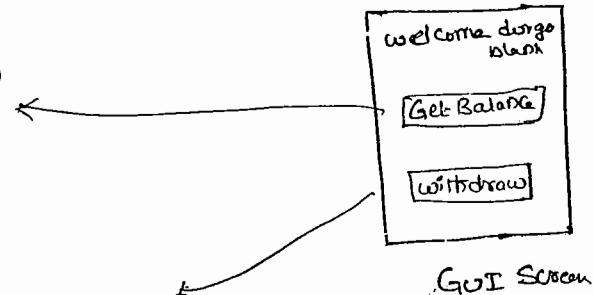
 {

 // validate user

 this.balance = balance;

 }

}



GUI Screen

→ Hiding data behind methods is the Central Concept of Encapsulation

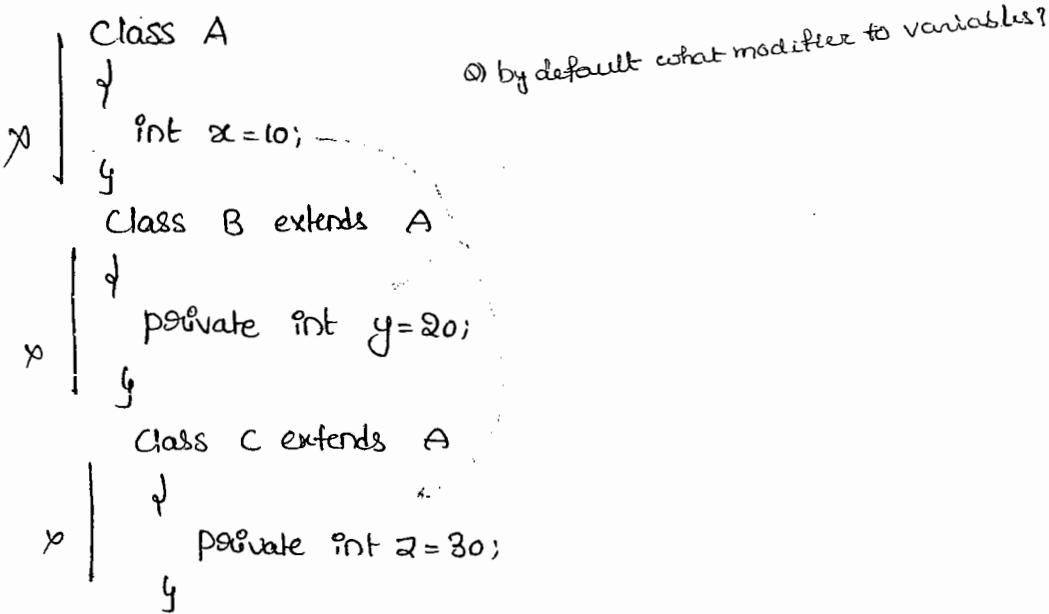
→ The main **advantages** of Encapsulation are ① We can achieve Security.

② Enhancement will become very easy.

③ Improves modularity of the application.

Ex 3) Which of the following classes are Tightly Encapsulated.

Ex:-



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)

↓

Case 1: 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(); ✓

* Case 3:

P p = new C();

p.m1(); ✓

P.p.m2(); X → c.e!

* Case 4:

C c = new P(); X c.e! incompatible types

found : P

required : C

Conclusion(1):

① whatever the parent class has by default available to the child. hence ^{with theon} child class reference we can call both parent & child class methods.

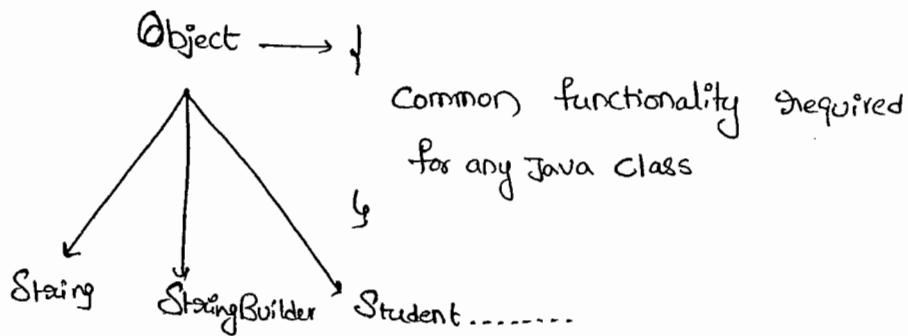
② whatever the child has by default not available to the parent hence on the parent class reference we can call only parent class methods & we can't call child specific methods.

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, its functionality will be available automatically to every child not required to rewrite.

Q) Does 'Throwable' have 'Object' as parent class?
Ans: Yes

→ Java won't provide support for **multiple inheritance** but through interfaces it is possible.

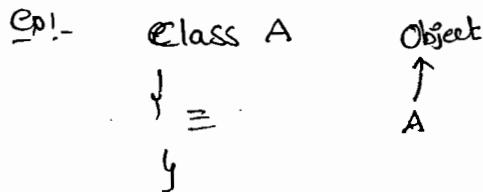
Ex:-

Class A extends B, C

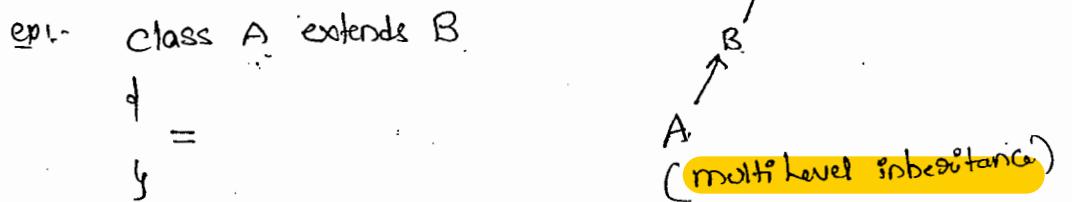
But interface A extends B, C

X Y
C.EI.

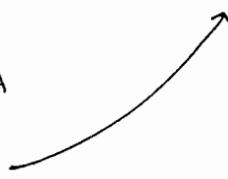
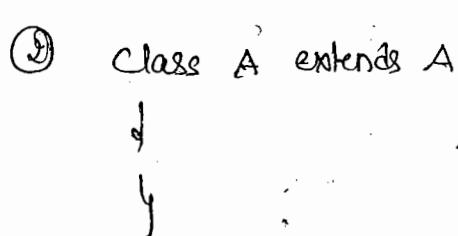
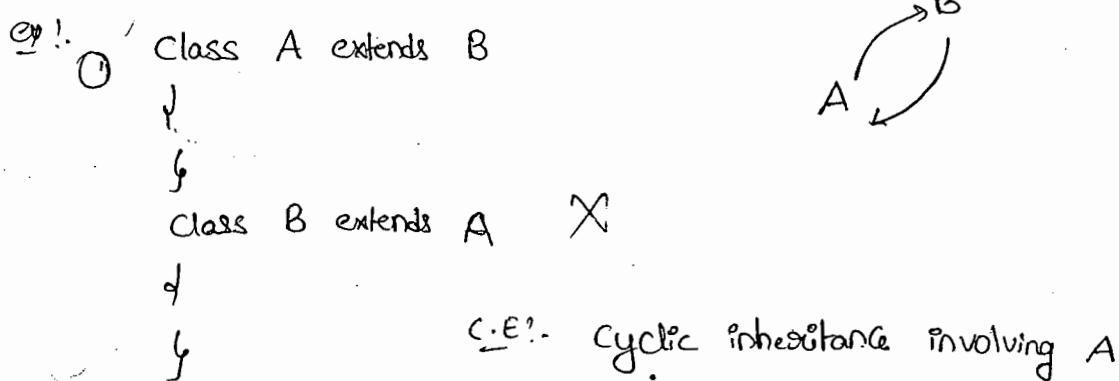
- Every class in Java is the child class of Object.
- If our class doesn't extend any other class then only it is the direct child class of Object.



- If our class extend any other class then our class is not directly child class of Object.



- Cyclic inheritance is not allowed in Java



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

↓

Engine e = new Engine();

{

Class Engine

↓

// Engine Specific functionality

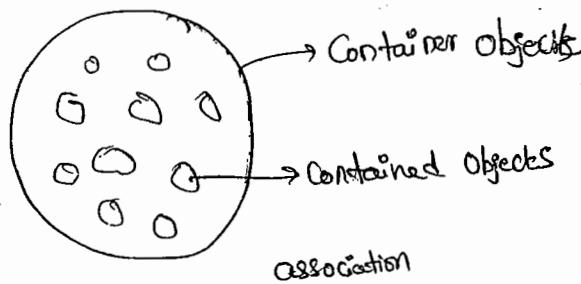
{

Class Car has Engine reference.

- The main **disAdvantage** of Has-A Relationship is it increases dependency b/w the classes and creates maintenance 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 b/w Container & Contained objects having Strong association



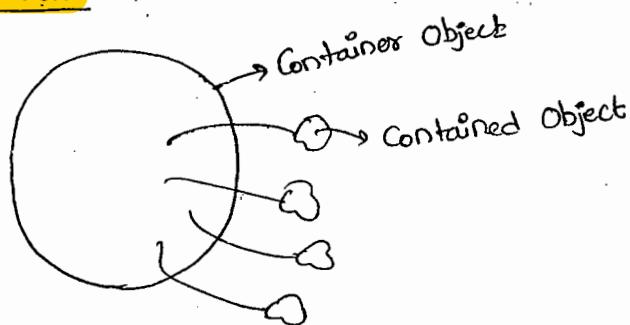
Ex:-

- 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 guarantee of destruction of Contained Objects ie, without existing Container Object there may be a chance of existing Contained Object i.e, Container Object just maintains References to 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)
{
    System.out.println("int-arg");
}

public void m1(float f)
{
    System.out.println("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.

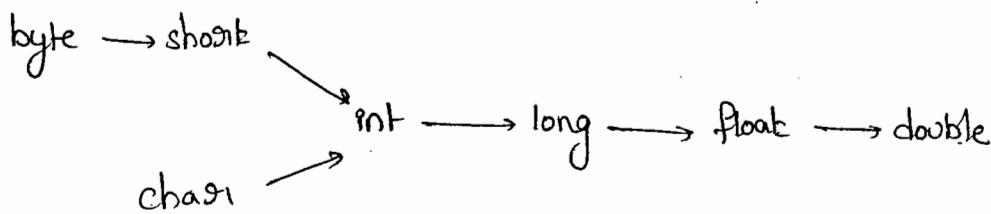
→ In Overloading reference type will play very important role & Runtime Object will be dummy.

Case 1 :-

* 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. If it promotes that assignment to the next level and checks for matched method.
- If the matched method is available then it will be considered and if it is not available then Compiler once again promoted this assignment to the next level.
 - This process will be continued until all possible promotions after completing all promotions still if the matched method is not available then only we will get C.E.
 - This is called **Automatic promotion** in overloading.
 - The following are various possible promotions in overloading.



Case 1 :-

Ex:- Class Test

```

public void m1(int i)
{
    System.out.println("int-arg");
}

public void m1(float f)
{
    System.out.println("float-arg");
}

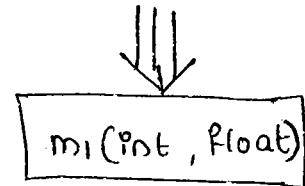
public void m1(String[] args)
{
}
  
```

Test t = new Test();

Method Signature :-

→ method Signature consists of name of the method & argument-list.

Eg:- public void m1 (int i, float p)



→ In Java return type is not part of method Signature.

→ Compiler will always use method Signature while resolving method calls

→ Within the same class Two methods with the same signature

not allowed. Otherwise we will get Compiletime Error.

Eg:- class Test

```
public void m1 (int i)
```

```
public int m1 (int i)
```

```
return 10;
```

m1 (int)
is the method Signature.

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

```
t.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 arguments are different.
- Lack of overloading in 'C' increases Complexity of the program.

In C, language if there is a change in method argument type

Compulsory 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 arguments 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()  
{  
    System.out.println(" no -arg");  
}
```

Case 1:- In Overloading mode more specific version will get highest priority.
what does it mean?

104

Case 2:-

Ex:- Class Test

↓
public void m1(StringBuffer sb)

↓
System.out.println("StringBuffer - args");

↓
public void m1(String s)

↓
System.out.println("String - version");

↓
public static void m1()

~~By default~~

By default 'String'
constant of String class
object type

t.m1(new StringBuffer("duaga")); //StringBuffer - args

t.m1("duaga"); // String version

X t.m1(null);

X // C.E. - reference m1() is ambiguity.

the integral constant of 'int'
floating literal "Hello"

t.m('a'); // int-arg

t.m(10); // float-arg

t.m(10.5); X c.e.

Cannot find symbol

Symbol: method m (double)

location: class Test

Case 2:-

→ In overloading method resolution child-argument will get more priority than parent argument.

Eg:-

Class Test

{

① public void m1(Object o)

{

System.out.println("Object Version");

}

② public void m1(String s)

{

System.out.println("String Version");

}

p. S.v.m (—)

{

Test t = new Test();

t.m1(new Object()); // Object-version

t.m1("String"); // String-version (String the obj is object)

t.m1(null); // String-version

Object

↑

String

Suppose ② statement takes //

→ 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 & assignments must be matched i.e., method signatures must be matched.
- ② In overriding return type must be matched, But this rule is applicable until 1.4 version, from 1.5 version onwards Co-variant 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()



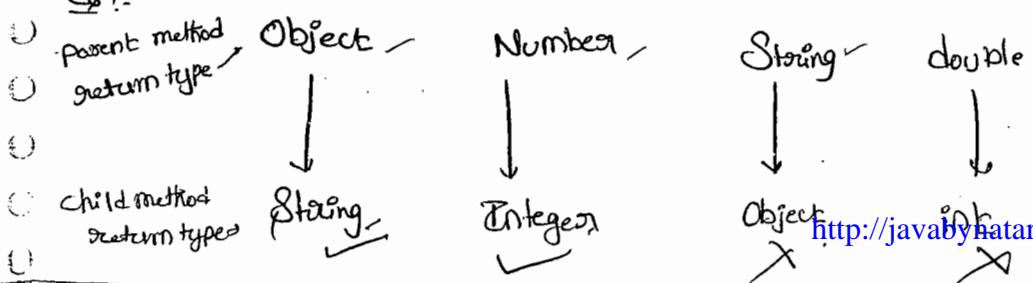
↳ return null;



It is valid in 1.5v,

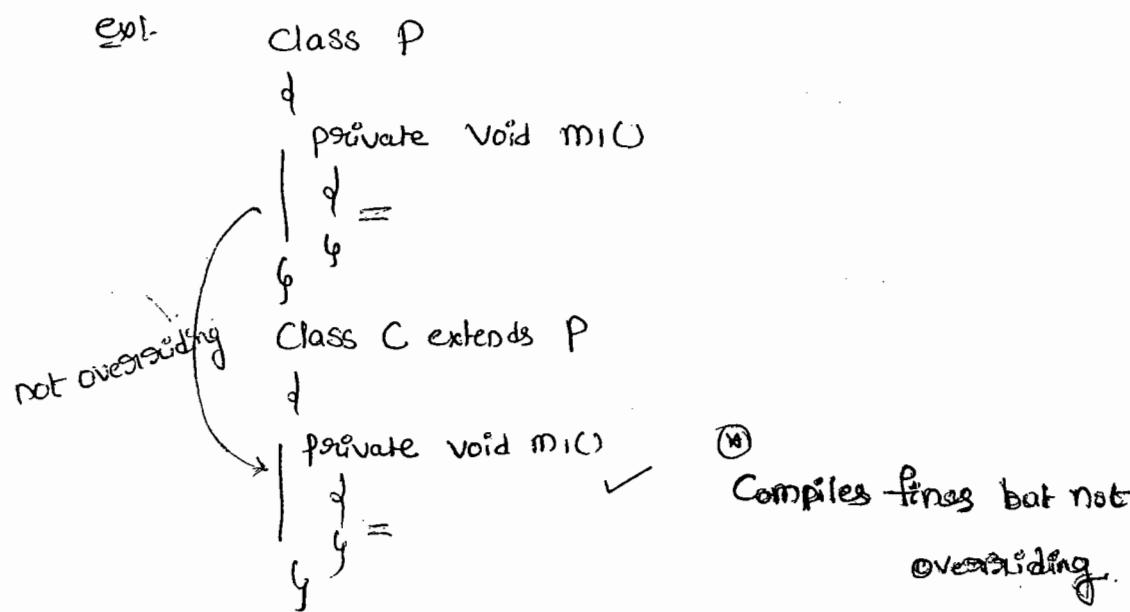
But invalid in 1.4v

Ex!:-



→ Co-variant 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 overriding 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.



→ 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

```

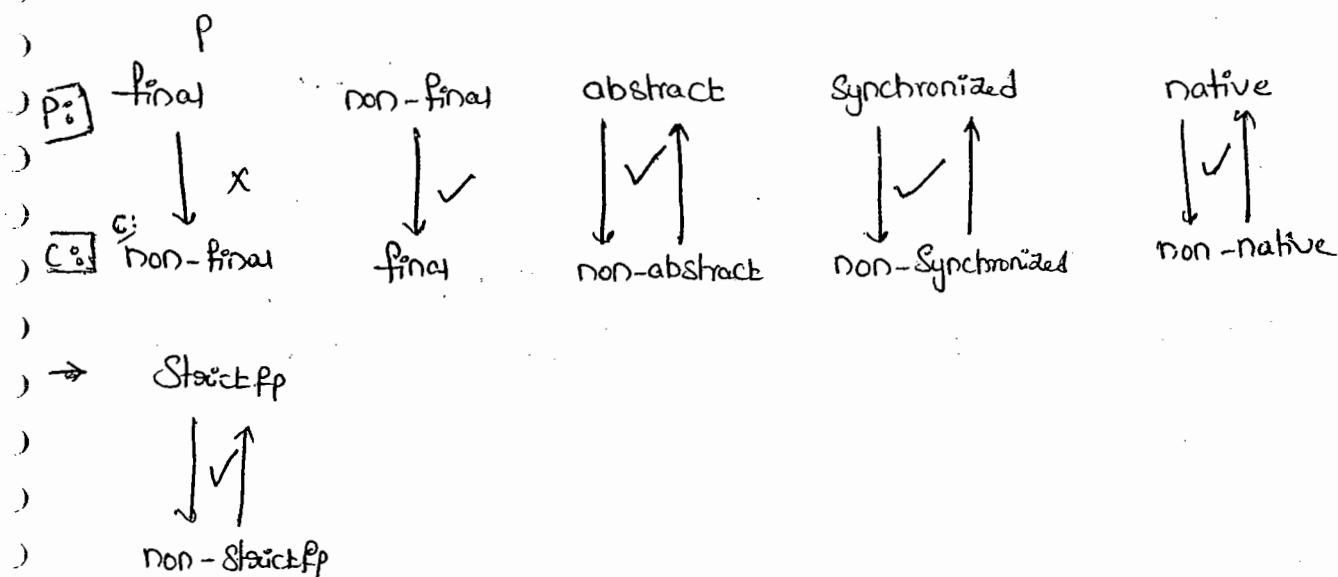
    |
    public abstract void p(); ✓
  
```

```

    |
    |
    |
  
```

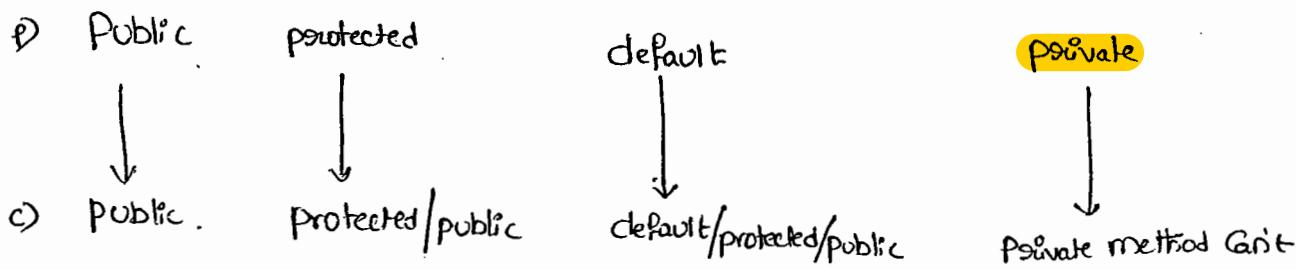
→ The following modifiers won't play any restrictions in overriding

- 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



Eg:- Class P

↓
public void m1() { }

}

Class C extends P

↓

protected void m1() X

}

}

C.E!

m1 in C Can't override in C

→ This rule is applicable while implementing interface methods also.

→ Whenever we are implementing any interface method Compulsory it should be declared as public. Because every interface method is public by default.

Ex:-

interface Interf

↓

void m1();

Y

Class Test implements Interf

if we declare
public we won't
get any C.E

↓

void m1()

{ } { }

X C.E:-

→ If child class method throws some checked exception then compulsory parent class method should throw the same checked exception or its class exception.
Parent, 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 X

↳

↳ ↳ C.E! - m1() in C can't override m1() in P;

Overridden method does not throw exception.

Ex②:-

Ⓐ P: public void m1() throws IOException

✓ C: public void m1()

Ⓑ P: public void m1()

X C: public void m1() throws IOException

Ⓒ P: public void m1() throws Exception

✓ C: public void m1() throws IOException

Ⓓ P: public void m1() throws IOException

X 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()
 }
 }

Static
↓
non-static

Class C extends P

{
 public void m1()
 }
 }

X

C.EI. m1() can't override m1() in P;

Overridden method is static.

→ Similarly, we can't override non-static method as static.

→ If both parent & child class method ~~class~~ are static then

We won't get any CE 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

- 1) Both methods should be static
- 2) Method Resolution takes care by Compiler based on Reference type.
- 3) It is Considered as Compile-time Polymorphism or Static Polymorphism or Early Binding.

Overriding

- 1) Both methods should be non-static.
- 2) Method Resolution always takes care by JVM based on Runtime Object.
- 3) It is Considered as Runtime Polymorphism or Dynamic Polymorphism or Late Binding.

Ex:-

Class P

 |
 | public static void m1()
 |
 | | System.out.println("parent");
 |
 |
 |

Class C extends P

 |
 | public static void m1()
 |
 | | System.out.println("child");
 |
 |
 |

method hiding

Class Test

 |
 | p.s.v.m1();
 |
 |

P p = new P();

p.m1(); → parent

C c = new C();

c.m1(); → child

P p1 = new C();

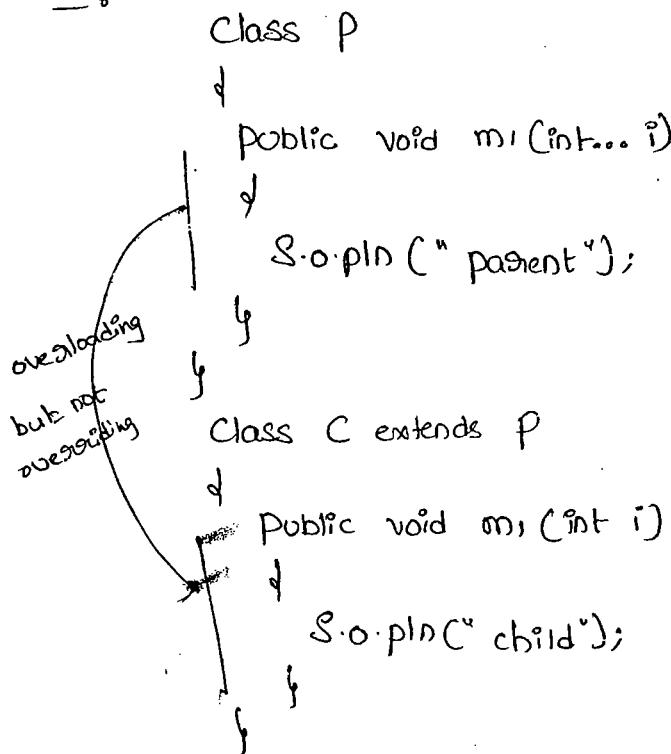
p1.m1(); Parent

→ If both methods are non-static then it will become overriding in this case the o/p is: Parent
Child
Child

Oversriding 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 **oversriding**.
- A Var-arg method should be overridden with Var-arg method only.

Ex:-



Class Test

```

    p.s.v.m(→)
    ↓
    P p = new P();
    p.m1(10); // Parent
    C c = new C();
    c.m1(10); // child
  
```

P p = new C();

P.m1(10); // Parent

→ If both parent & child class methods are Var-a-ing Then it will becomes overriding in this case o/p is parent child child

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 play any role in variable resolution.

Ex:-

Class P

↓
int x=888;

both static

Class C extends P

↓
int x=999;

↓

Class Test

↓

P·S·V·M(→)

↓

P p =new P();

S.o.println(p.x); // 888 ↗

C c =new C();

S.o.println(c.x); // 999 ✓

P p1 =new C();

S.o.println(p1.x); 888.

↓
}

both static	both instance	one static & one instance
o/p 888	o/p 888	o/p 888

→ whether 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 arguments	must be different (at least order)	must be same (including order)
③ Method Signature	must be different	must be same.
④ Return type	No restrictions	must be same until 1.4v but from 1.5v onwards Co-varient return types are allowed.
⑤ private, static & final methods	Can be overloaded	Can't be overridden
⑥ access modifiers	No restrictions	Scope we can't decrease the scope.
⑦ throws clause	No restrictions	Size & level of checked exceptions we can't increase but we can decrease. But no restrictions 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) & arguments (must be diff.) All remaining terms like return type, throws clause, access modifiers etc. 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

- ① public void m1(int i) *Overriding*
- ② public void m1() throws Exception *overloading*
- ③ public static int m1(double d) throws IOException *overloading*
- C.E X ④ public int m1(int i)
- C.E X ⑤ public synchronized void m1(int i) throws Exception
- overloading ⑥ public static void m1(int... i) throws Exception
- C.E X 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.

Polymorphism

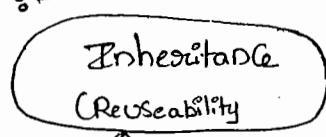
Compile-time polymorphism

Ex:- Overloading
Method Hiding

Run-time polymorphism

Ex:- Overriding.

3 Pillars of OOPS :-



OOPS

Encapsulation
(Security)

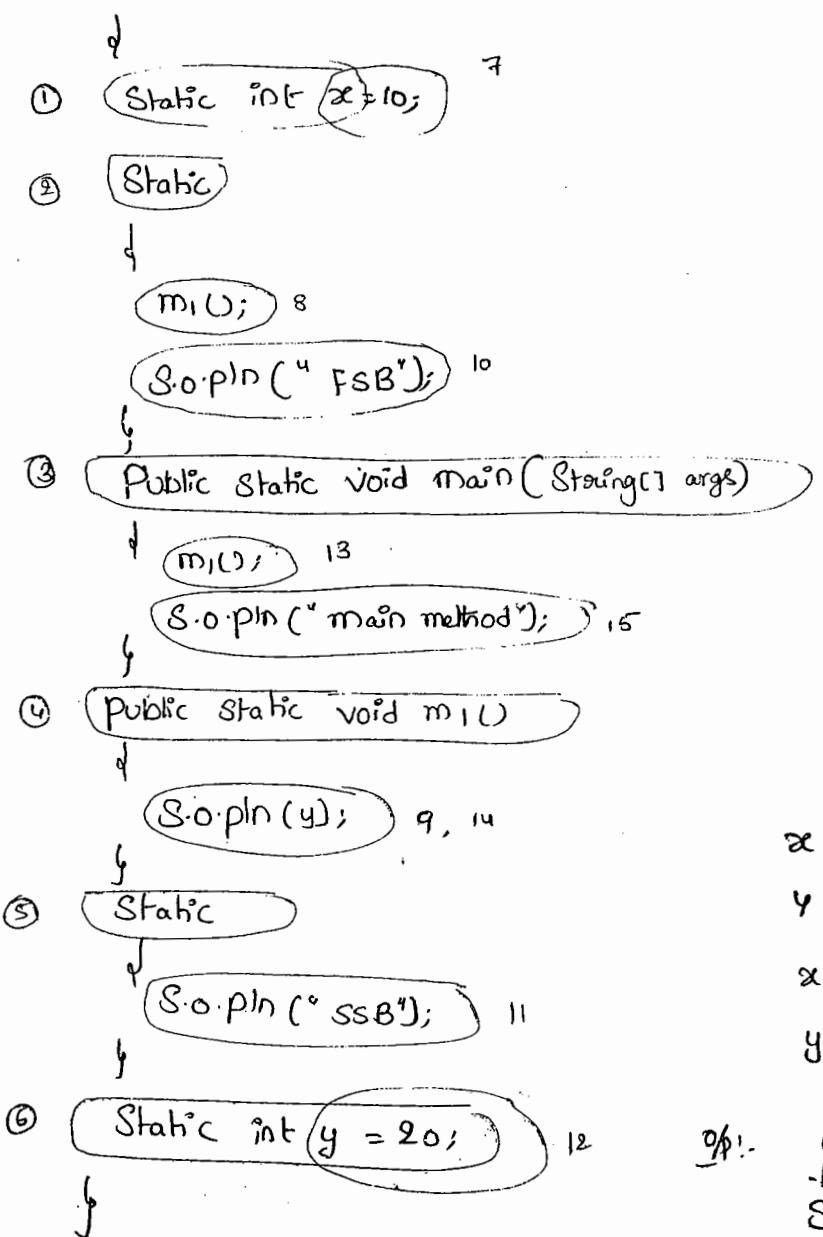
Polymorphism
(Flexibility)

Funny definition of polymorphism :-

→ 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 polymorphism.

Static Control flow :-

Ex:- Class Base



x = 0 [R I W]

y = 0 [R I W]

x = 10 [R & W]

y = 20 [R & W]

O
FSB
SSB
20

main method

Process:-

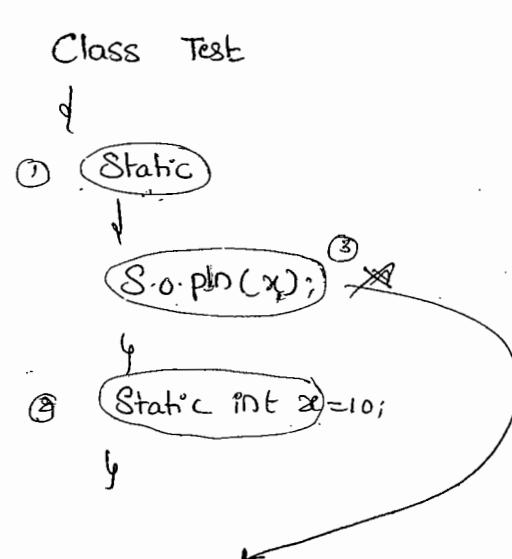
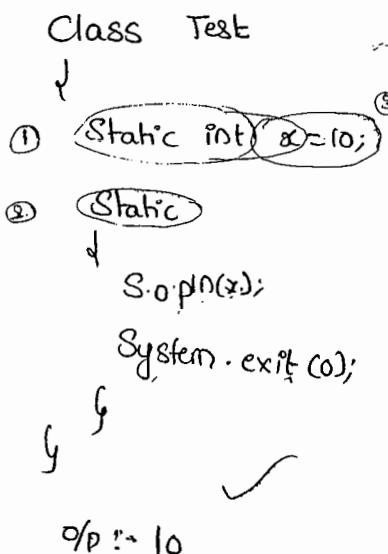
→ When ever we are trying to execute a Java class first that .class file should be loaded, at the time Class loading The following actions will be performed automatically.

- 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".

१०



C.E.:- Illegal forward reference.

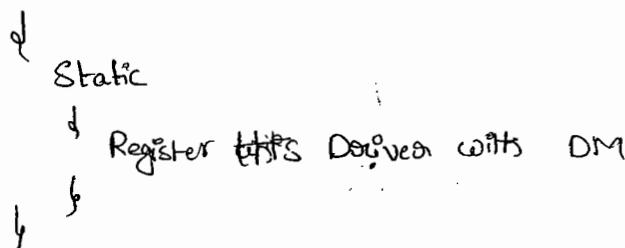
Static block :-

- 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.
- Within a class we can take any no. of 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 driver manager but every Driver class contains a static block to perform this activity at the time of Driver class loading automatically we are not responsible to perform register explicitly.

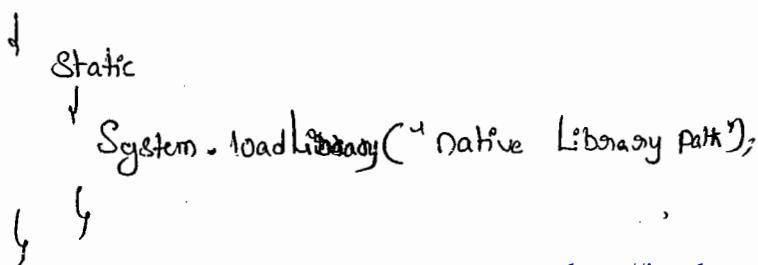
Ex:- Class Driver



Ex(2) :- Advantage:

- At the time of class loading ^{Compulsory} we have to load the corresponding native libraries hence we can define this step inside static block.

Ex:- Class Native



Static Control flow in parents child classes :-

Class Base

{

① Static int x=10; ⑫

② Static

{

m1(); ⑬

{

S.o.println(" Base SB"); ⑭

③ Public static void main(—)

{

m1();

{

S.o.println(" Base main");

{

④ public static void m1()

{

S.o.println(y); ⑮

{

⑤ Static int y=20; ⑯

{

Class Derived extends Base

{

⑥ Static int j=100; ⑰

⑦ Static

{

m2(); ⑱

{

S.o.println(" DFSB"); ⑲

③ Public static void main(—)

{

m2(); ⑳

{

S.o.println(" Derived main"); ㉑

⑨ Public static void main()

↓

System.out.println(j); ⑯ ⑰

↓

⑩ Static

↓

System.out.println("DSSB"); ⑱

↓

⑪ Static int j=200; ⑲

↓

> Java Derived

O/P:- 0

Base SB

0

DSSB

DSSB

200

Derived main

x=0 [R I WO]

y=0 [R I WO]

i=0 [R I WO]

j=0 [R I WO]

x=10 [R I W]

y=20 [R & W]

i=100 [R & W]

j=200 [R & W]

> Java Base

0

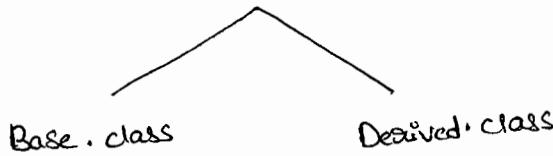
Base SB

20

Base main

Process:-

> javac Derived.java



> java Derived

① Identification of static members 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 m(); ⑩

System.out.println("FIB"); ⑫

}

⑤ Parent()

{

System.out.println("Constructor"); ⑯

}

① Public static void main(String args)

{

② Parent p = new Parent();

System.out.println("main");

}

⑥ Public void m()

{

System.out.println(y); ⑪

}

⑦ { → instance block

System.out.println("SIIIB"); ⑬

}

⑧ int y=20; ⑭

}

x=0 [R I W O]

y=0 [R I W O]

x=10 [R W]

y=20 [R W]

O/P:-

O

FIB

SIIIB

Constructor
main

Process :-

→ whenever we are creating an object the following sequence of events will be performed automatically.

- (1) Identification of instance members from top to bottom [1 to 8]
- (2) Execution of instance variable assignments & instance blocks from top to bottom [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; ⑯

④ ↓

m1(); ⑯

S.o.println("parent"); ⑯

}

⑤ Parent()

↓

S.o.println("parent constructor"); ⑯

}

① public static void main()

↓

② Parent p = new Parent();

System.out.println(" Child main"); ③
y

④ public void m1()

↓

System.out.println(y); ⑤

y

⑥ int y=20; ⑦

y

Class Child extends Parent

y

⑧ int i=100; ⑨

y

⑩ m2(); ⑪

System.out.println(" CIIIB"); ⑫

y

⑬ Child()

y

System.out.println(" Child Constructor"); ⑬

y

⑭ Public static void main()

y

⑮ Child c = new Child();

y

System.out.println(" Child main"); ⑯

y

⑰ public void m2()

y

System.out.println(j); ⑰

O

Parent

Parent Constructor

O

CIIIB

CSIIIB

Child Constructor

Child main.

113

93

(15) S.O.P.I.N ("CSIIB"); (26)
↳
(16) $\int = 200$; (27)

Process :-

- whenever we are creating child class object
the following sequence of events will be performed automatically.
 - (1) Identification of instance members 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 Compulsory we should perform initialization Then only that Object is in a position to provide response properly.
- When ever we are creating an object Some part of the code will be executed automatically to perform initialization, this part of code is nothing but constructor. Hence the main objective of constructor is to perform initialization for the newly created object.

Q:-

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)

{

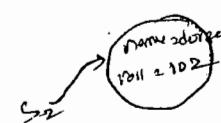
Student s₁ = new Student ("durga", 101);

Student s₂ = new Student ("raghu", 102);

}

}

O
Object null.



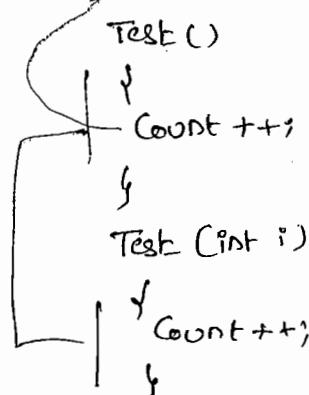
Instance block vs Constructor :-

- 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 constructor with instance block because constructor can take argument whereas instance block can't take arguments.
- 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 maintenance problems.

Ex:- class Test

 ↓
 Static int Count = 0;

Only once required



p.s.v.m(—)

↓
Test t₁ = new Test();

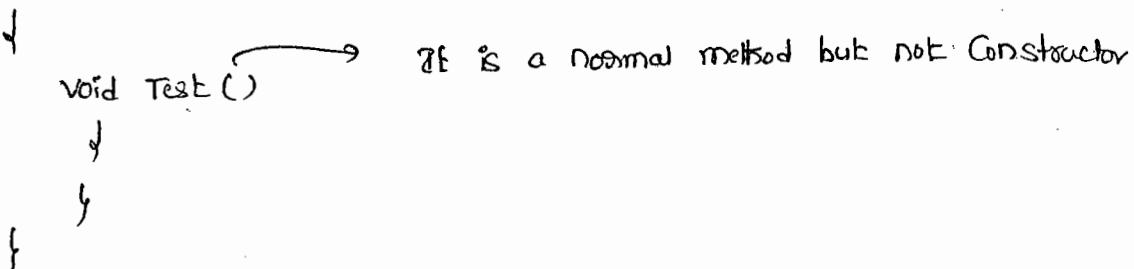
{ } Test t₂ = new Test();

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 won't get any Compile-time (or) Runtime Errors, because Compiler treats it as method.

Ex:- Class Test

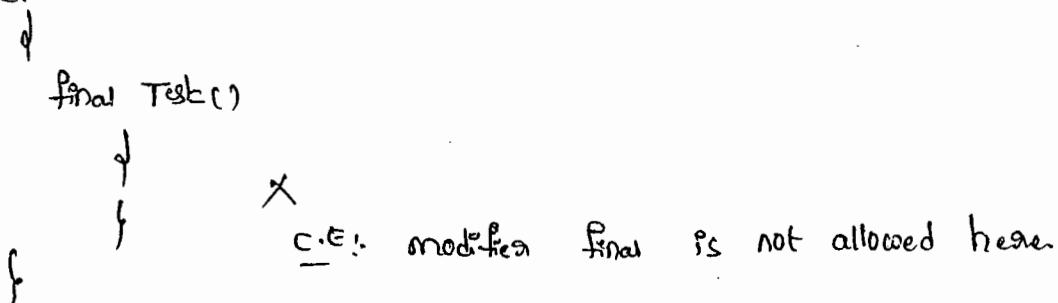


It is legal (for stupid) to have a method whose name is exactly same as class name).

- (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 / Statically - -

Ex:- Class Test



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 (Struts i.x)

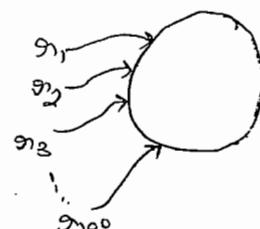
BusinessDelegate (EJB), ServiceLocator (EJB) ---- etc.

→ 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 $s_1 = \text{Runtime.getRuntime}()$

Runtime $s_2 = \text{Runtime.getRuntime}()$
 ↓
 Class Static method

Runtime $s_{100} = \text{Runtime.getRuntime}()$



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()

{

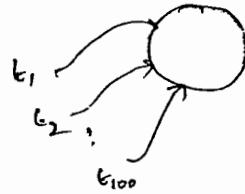
```

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 r₁ = Runtime.getRuntime();

↗ factory method

Dateformat df = DateFormat.getInstance();

↗ factory method

Test t = Test.getInstance();

↗ factory method

12D
96

→ Similarly we can Create Doubleton, Tripleton ----- 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 written constructor or compiler generated constructor but not both simultaneously.

Prototype of Default Constructor :-

- 1) It is always no argument 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 argument call to Super class constructor.

```
Test()  
↓  
Super();  
↳
```

Programmer's Code

Compiler generated Code

121
91

(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

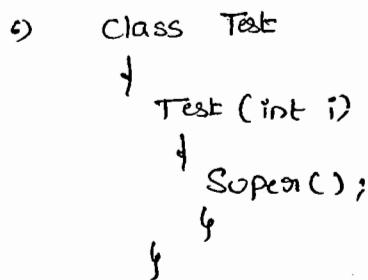
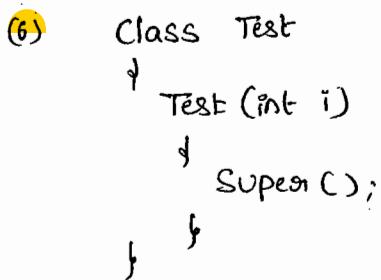
```
↓  
Test()  
↓  
Super();  
↓
```

(5) Class Test

```
↓  
Test()  
↓  
this(10);  
↓  
Test(int i)  
↓  
{  
}
```

(5) Class Test

```
↓  
Test()  
↓  
this(10);  
↓  
Test(int i)  
↓  
Super();  
↓
```

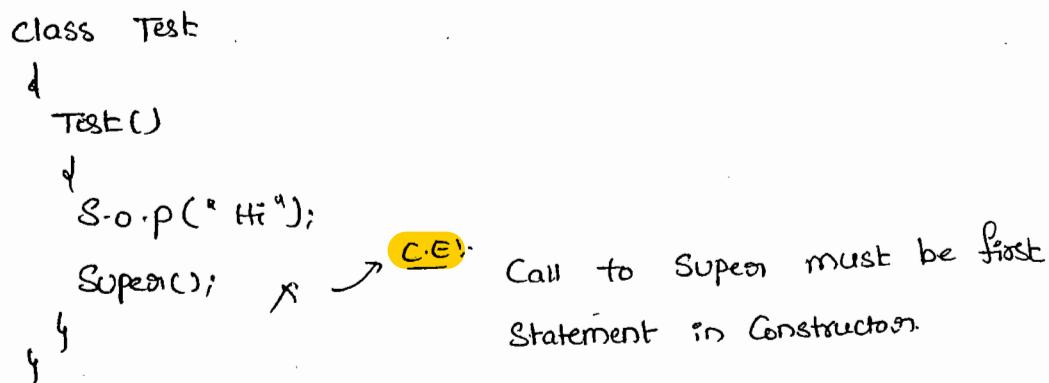


Super & This :-

- The first Line inside a Constructor should be either Super() or this().
- If we are not calling anything Compiler will always places Super().

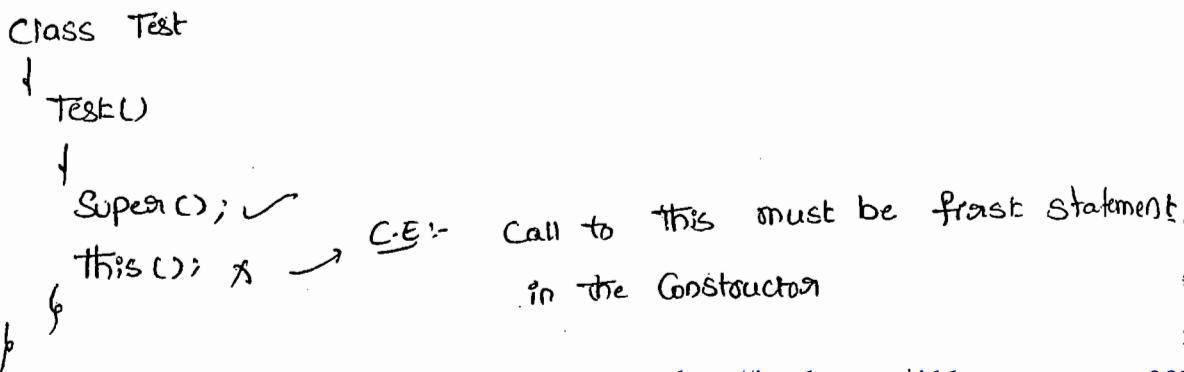
Case(i) :-

we have to keep either Super() or this() only as the first Line of the Constructor.



Case(ii) :-

Within the Constructor we can use either Super() or this() but not both simultaneously.



Case(iii) :-

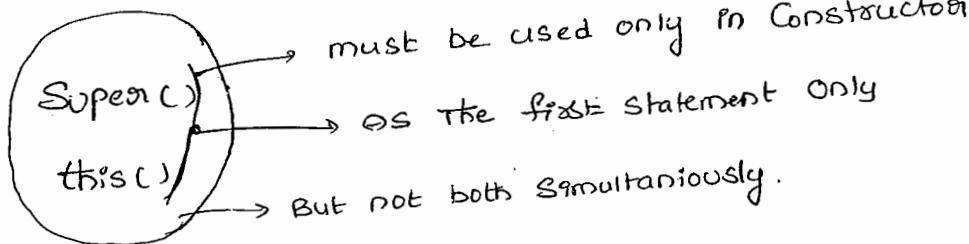
→ we can use Super & this only inside Constructor if we are using any where else we will get Compiletime error.

Ex:- Class Test

```
public void m1()
```

Super(); X → C.E:- Call to Super must be first statement in the constructor

```
System.out("Hi");
```



this() :- To Call Current class Constructors

Super() :- To Call Parent class Constructors

Compiler provides default Super() but not this().

Super()	Super this
<ul style="list-style-type: none"> (i) These are Constructor calls (ii) we should use only in Constructors 	<ul style="list-style-type: none"> i) These are key words to refer Super & Current class instance members ii) we can use any where Except in static area.

Ex:-

Class Test

|
P.S.V.m()

|
S.o.pn(Super.hashCode()); X

|
LCE! - 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 arguments & these constructors are considered as overloaded constructors.

Ex:-

Class Test

|
Test(double d)

|
this(10);

S.o.pn(" double-args");

|

Test(int i)

|
this();

S.o.pn(" int-args");

|

Test()

|
S.o.p(" No-args");

|

P.S.V.m(—)

|

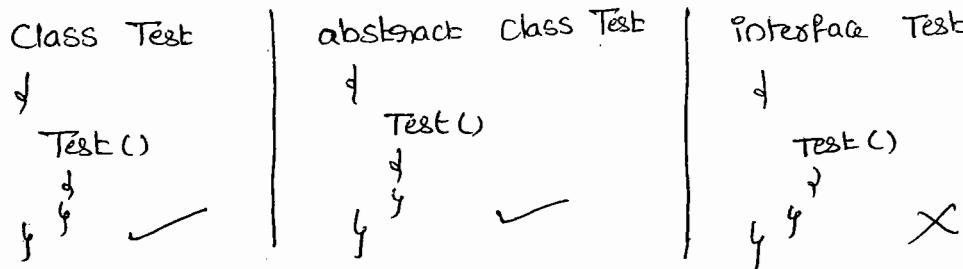
Test t₁ = new Test(10.5); → No-args
 int-args
 double-args

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 interface can't have the constructors.



→ Case(i):-

→ Recursive method call is always Runtime Exception whereas Recursive Constructor invocation is a Compiletime Error.

e.g:-

Class Test

|
P.S.V.m₁()

|
m₂();

|
P.S.V.m₂()

|
m₁();

|
P.S.V.m(→)

|
S.O.P("Hello");

|
m₁()) AE: Stack overflow Error

Class Test

|
Test()

|
this(10);

|
Test(int i)

|
this();

P.S.V.m(→)

|
S.O.P("Hello");

C.E: Recursive Constructor
<http://javabynataraj.blogspot.com>
 invocation.

Case(ii) :-

e.g. Class P



Class C extends P



Class P



p()



Class C extends P



Class P



p(int i)



Class C extends P



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.

→ whenever we are writing any argument Constructor it is highly recommended to write no argument Constructor also.

Case(iii) :-

→ If parent class constructor throws some checked Exception Compulsory Child class constructor should throw same checked exception or its parent otherwise the code won't compile.

class P



p() throws IOException



class C extends P

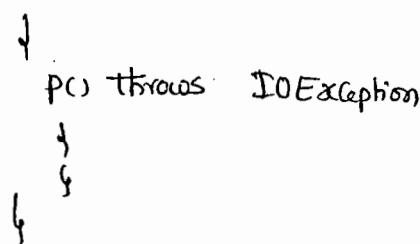


C.E:- unsupported exception Java.io.

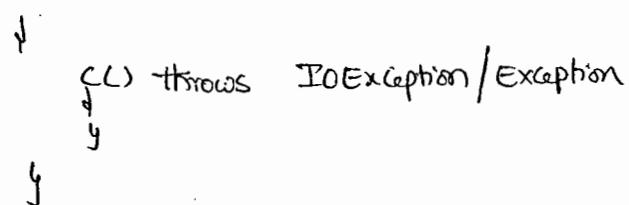
<http://javabynataraj.blogspot.com>

241 of 255.

Ex:- Class P



Class C extends P



Q) Which of the following is True?

- ① Every class Contains Constructors ✓
- ② Only Concrete Classes Can Contains Constructors but not abstract classes X
- ③ The name of the Constructor need not be same as class name X
- ④ Return type is applicable for the Constructor X
- ⑤ The only applicable modifiers for Constructors are public & default X
- ⑥ If we are trying to declare Return type for the Constructor we will get Compile-time Error X.
- ⑦ Compiler will always generate default Constructor X
- ⑧ The access modifier of the default Constructor is always default X
- ⑨ The first Line inside every Constructor should be Super X.
- ⑩ The first Line " " Should be Super(this), ✓
- If we are not writing anything compiler will always place this, X

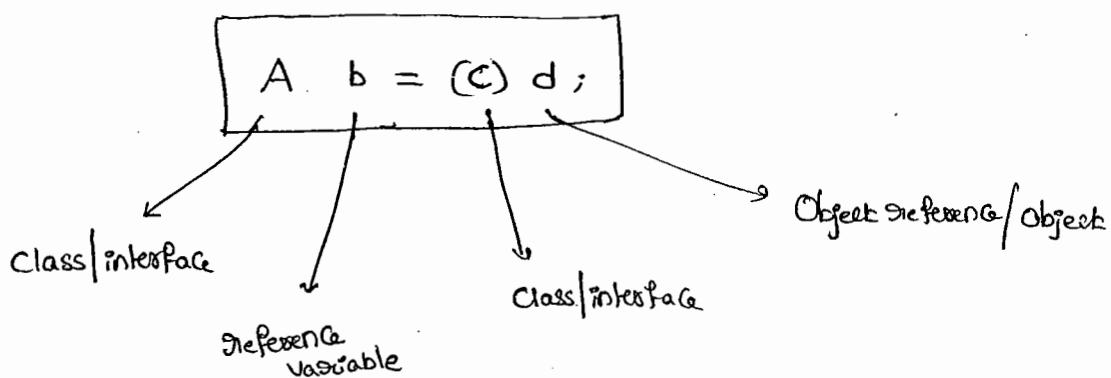
- (i) Interface Can Contains Constructor X
- (ii) Both overloading & overriding Concepts are applicable for Constructors X.
- (iii) Inheritance Concept is applicable for Constructors X

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(i) :-

- C & type of d must have Some relationship (either parent to child or child → parent or Same type) otherwise we will get Compilation Error saying "inconvertible-types - found d type but required C type".

Ex(1) :-

```
Object o = new String("durga");
```

```
StringBuffer sb = (StringBuffer) o;
```

Ex(2) :-

```
String s = new String("durga");
```

```
SB sb = (SB)s; X
```

C.E!:-

inconvertable types

found : java.lang.String

required : java.lang.SB

Compiler checking rule 2 :-

- C must be either same or derived type of A otherwise we will get Compiler time Error saying "incompatible 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 underlying object type of 'a' must be either same or derived type of C, otherwise we will get runtime exception saying "ClassCastException".

Ex:-

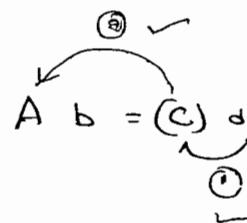
① Object o = new String ("durga");

SB sb = (SB) o; X

Rule ① ✓

② ✓

③ X (R-E):- CCE

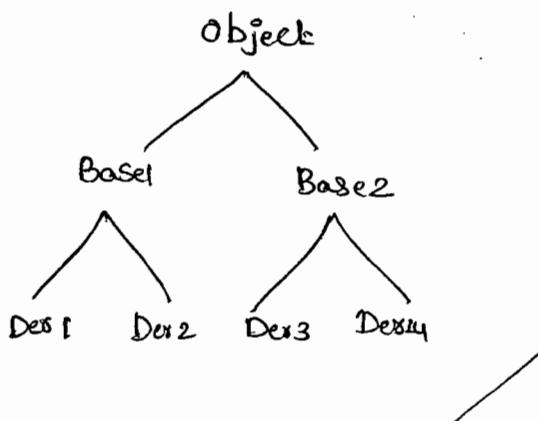


② Object o = new String ("durga");

String s = (String)o; ✓

Rule ① ✓
② ✓
③ ✓

Ex:-



C.E! - Inconvertible types

found : Base2

Required : Base1

Ex. ① Base2 b = new Der4();

✓ ② Object o = (Base2) b;

X ③ Object o = (Base1) b;

④ Base2 b1 = (Base2) o; → o is null

X ⑤ Base1 b3 = (Der1)(new Der2());

C.E!

✗ ⑥ Convertible types

found : Der2

Required : Der1

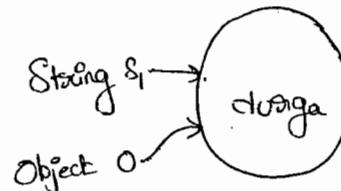
→ Strictly Speaking in type-casting just we are Converting only type to object but not underlying object itself

Ex:-

```
String s1 = new String("duaga");
```

```
Object o = (Object) s1;
```

```
s.o.println(s1 == o); true
```



Ex:-

```
A -----> public void m1()
  |
  |
  |-----> s.o.println("A");
  |
B -----> public void m1()
  |
  |
  |-----> s.o.println("B");
  |
C -----> public void m1()
  |
  |
  |-----> s.o.println("C");
  |
```

```
C c = new C();
```

```
c.m1(); → c ↗
```

```
((B)c).m1(); → c ↗ → B b = new C();
  |
  |-----> b.m1();
```

```
((A)c).m1(); → c ↗
```

```
→ A a = new C();
```

```
a.m1();
```

Eg 2:

```

A → public static void m() {
    ↑
    ↓
    System.out.println("A");
}

B → public static void m() {
    ↑
    ↓
    System.out.println("B");
}

C → public static void m() {
    ↑
    ↓
    System.out.println("C");
}

```

→ C c = new C();

c.m(); // C

→ ((B)c).m(); // B

→ ((A)c).m(); // A

Eg 3:

A → int x = 777;



B → int x = 888;



C → int x = 999;

C c = new C();

System.out.println(c.x); 999

System.out.println(((B)c).x); 888

System.out.println(((A)((B)c)).x); 777

(because The overriding Concept is not applicable)

→ If we declare all variables as static then there is no chance to change the o/p.

Note:-

→ whether the variable is static or instance variable resolution should be done based on preference 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.m();

Class C

↓
P.S.V.int m();

↓
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 ^{surrounding} any component we can't modify any component's <http://javabymraj.blogspot.com> 248 of 255.

Hence, enhancement will become difficult.

→ It reduces maintainability.

→ It doesn't promote reusability.

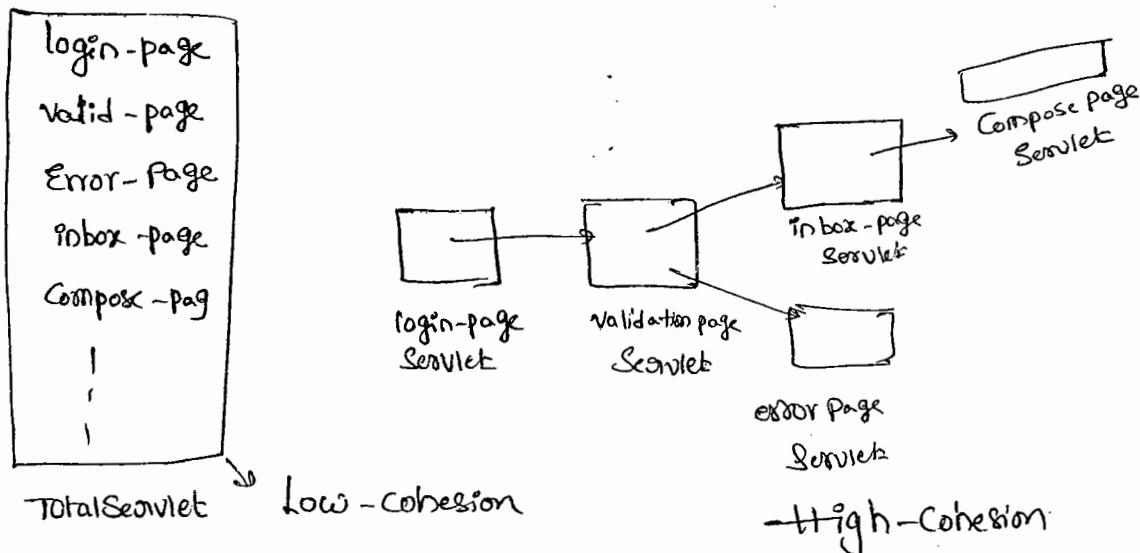
→ Hence it is highly recommended to maintain loosely coupling & dependency b/w the components should be as less as possible.

Cohesion

Cohesion :-

→ For every component a clear well-defined functionality we have to define, such type of component is said to be follow high-cohesion.

Ex:-



→ High-cohesion is always a good programming practice which has several advantages.

- (1) With out affecting remaining components we can modify any component hence enhancement will become very easy.

- (2) It improves maintainability of the application
- (3) It promotes reusability of the code.

Ex:-

→ where ever validation is required we can reuse the same validate Servlet without modifying.

Note:-

Loosely Coupling & high-cohesion are good programming practices.

=====
====

This:

To use the current class reference

means without creating multiple objects, ~~only~~ only one object is created then
call those values from the current class.

Kondalu-7@yahoo.co.in

23, 2

① Collection Framework (1 - 43)

Collection (6 - 24)

Map (25 - 37)

Collections class (38 - 40)

ArrayLists class (41 - 43)

② Generics (44 - 52)

③ Multithreading (53 - 76)

Synchronization (67 - 76)

④

Re

⑤ Regular Expressions (77 - 82)

⑥ Enumeration (83 - 89)

⑦ I18N (90 - 95)

⑧ Development (96 - 101)

