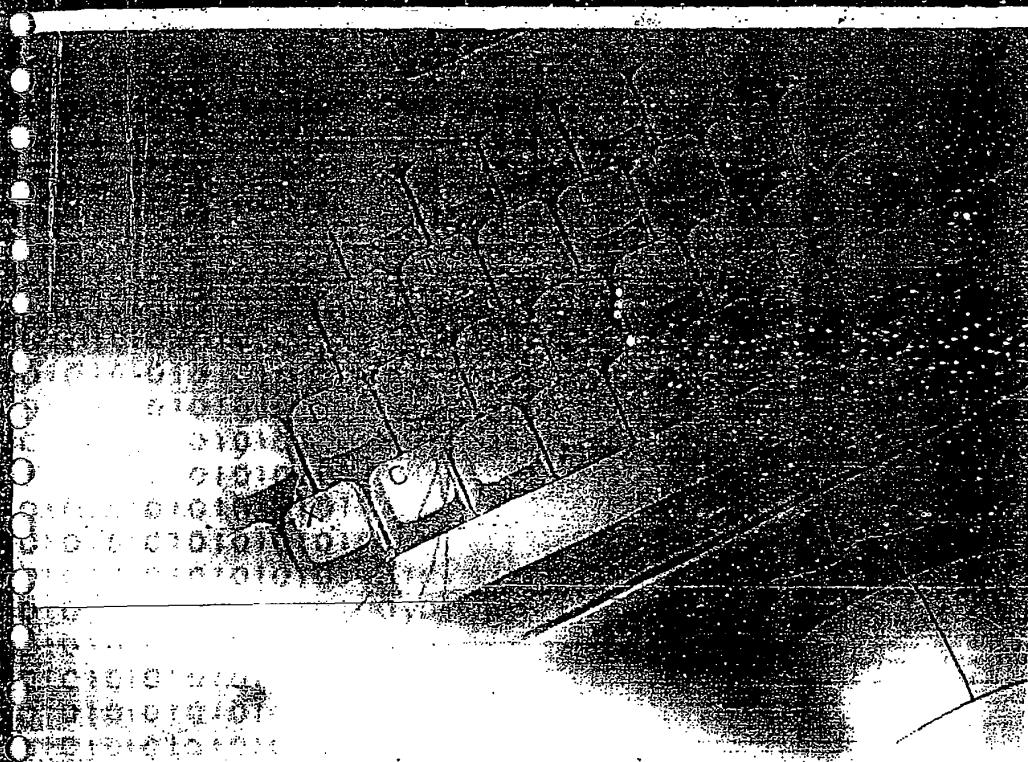


Learning Java is not enough...
Be Certified Professionals.



Java

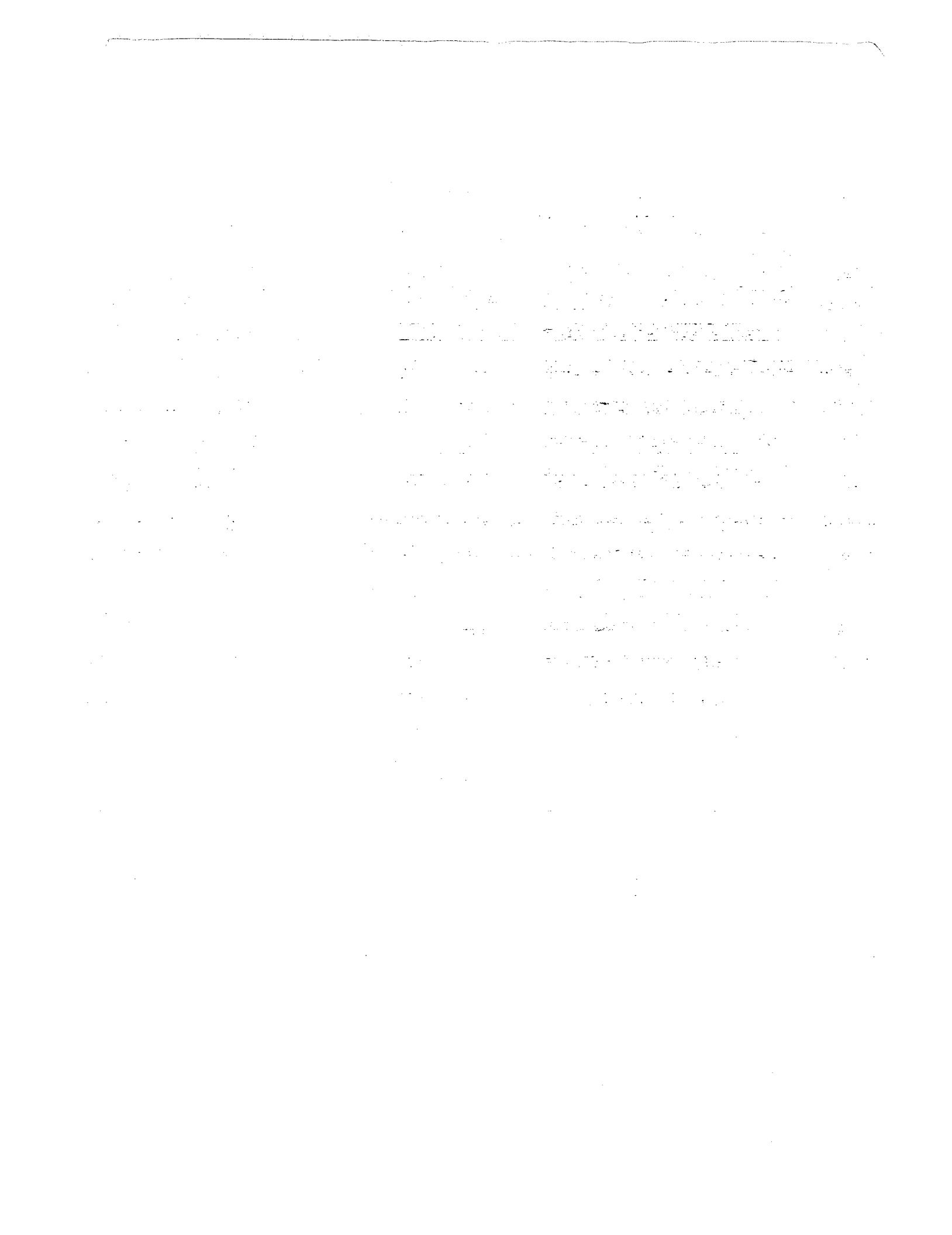
SCJP

Sun Certified
Java Programmer

SCWCD

Sun Certified
Web Component Developer

Software Solutions®



21/8/11

Language Fundamentals

① Identifiers

② Reserved Words 3

③ Data types 5

④ Literals 9

⑤ Arrays 13

⑥ Types of Variables 22

⑦ Var-arg methods 28 (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 {
 public static void main(String args) {
 int x=10;
 }
}`

 ↓ ↓ ↓
 classname method name variable name
 ↓ ↓ ↓
 p - s - v. main (String args)
 ↓ ↓ ↓
 int x = 10;
 ↓ ↓
 variable name ✓ → is identifier.

* 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 Compiletime Error.

Ex:- ✓ all-member
 ✗ all#
 ✓ -\$-\$
 ✗ 098\$-10

2) Identifier can't start 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 lengths (> 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's not recommended.

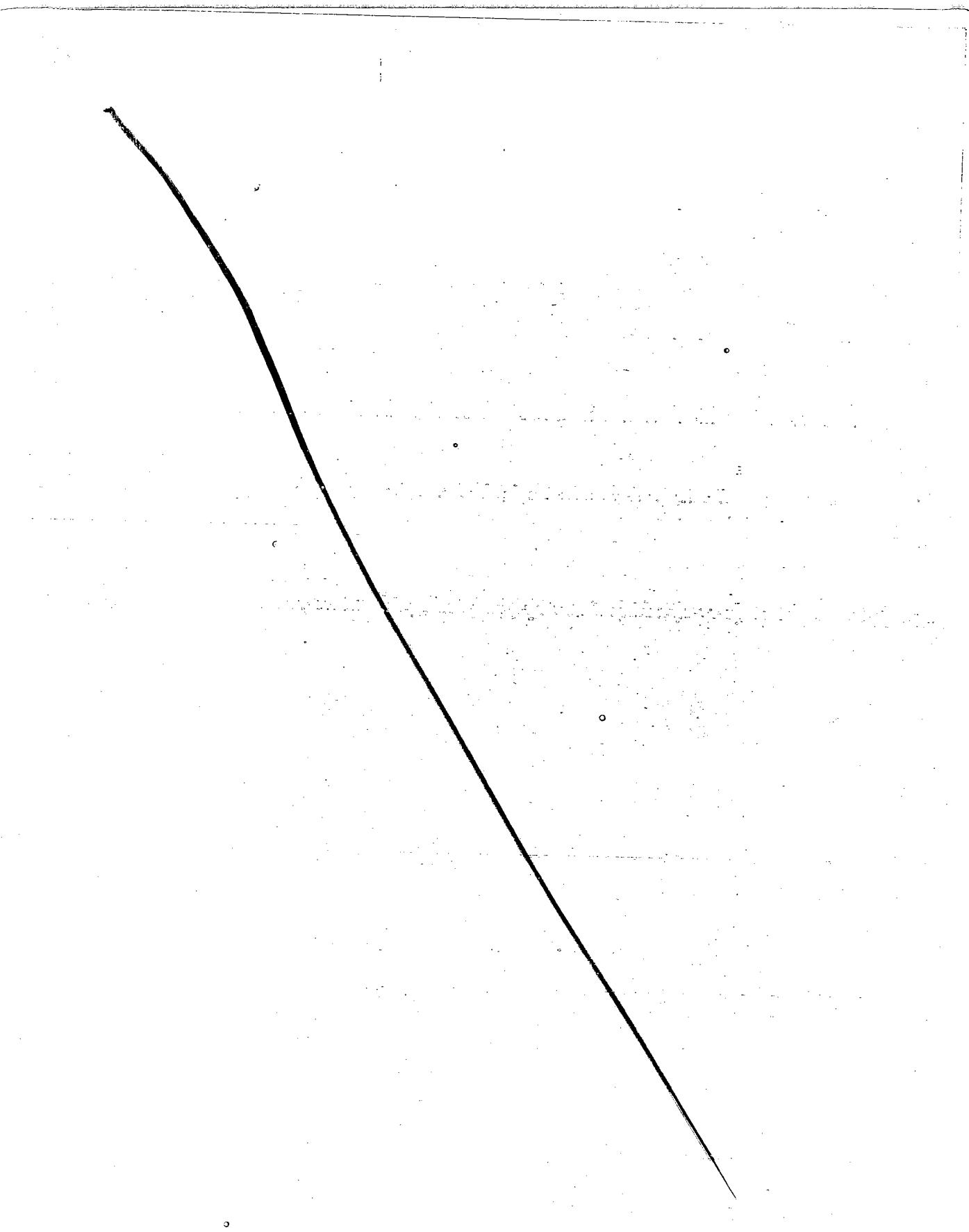
Ex:-

```
class Test
{
    int String = 10;
    System.out.println(String); 10
}
```

```
class Test
{
    int Runnable = 20;
    System.out.println(Runnable); 20
}
```

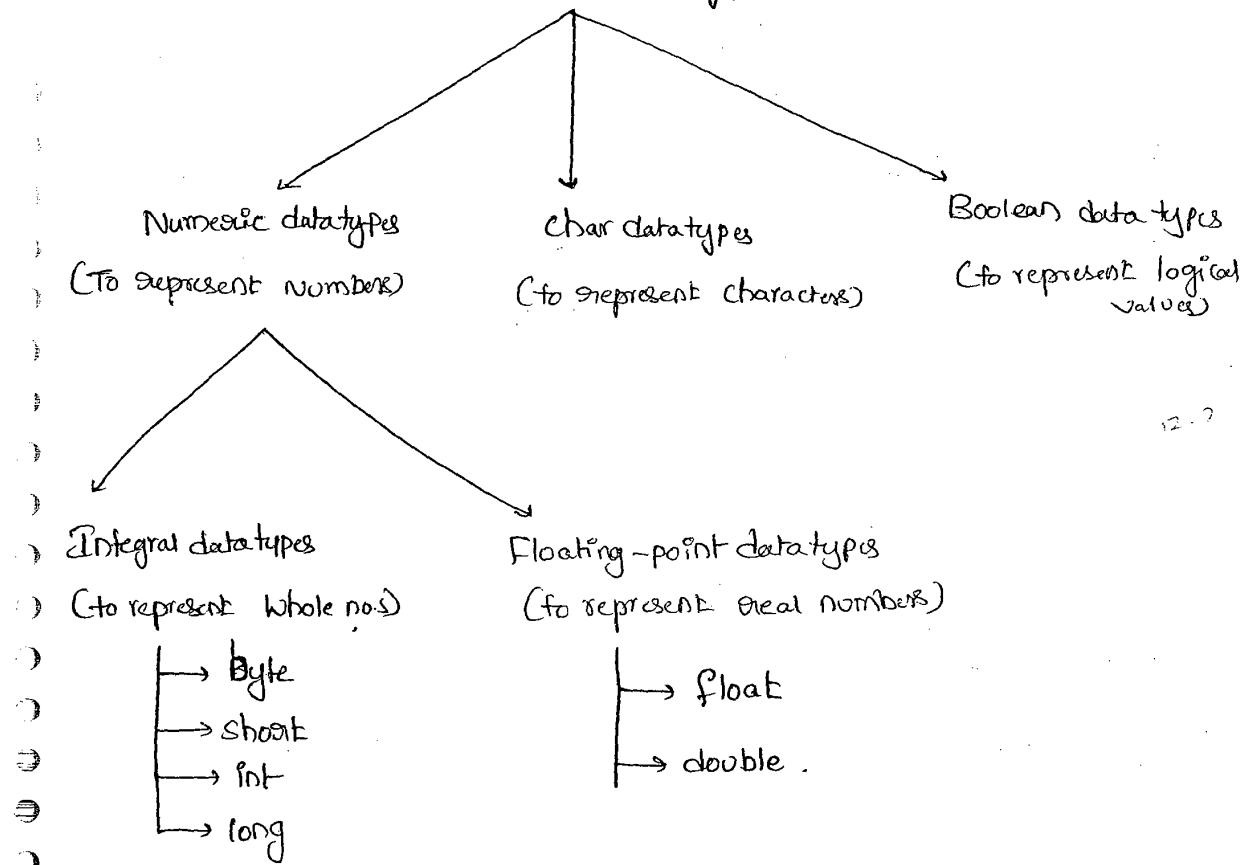
Q) Which ~~the~~ following are valid Java identifiers?

- ① Java2shape
- X ② 4shared
- X ③ all@hands
- ④ total-not-Students
- ⑤ -\$-
- X ⑥ total#
- X ⑦ int
- ⑧ Integer



4

Primitive data types (8)



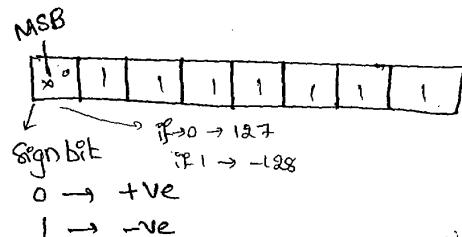
① Byte :-

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~~. lang. String

Required : byte.

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

② short :-

Size : 2-bytes (16-bits)

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

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

Eg! ✓ 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

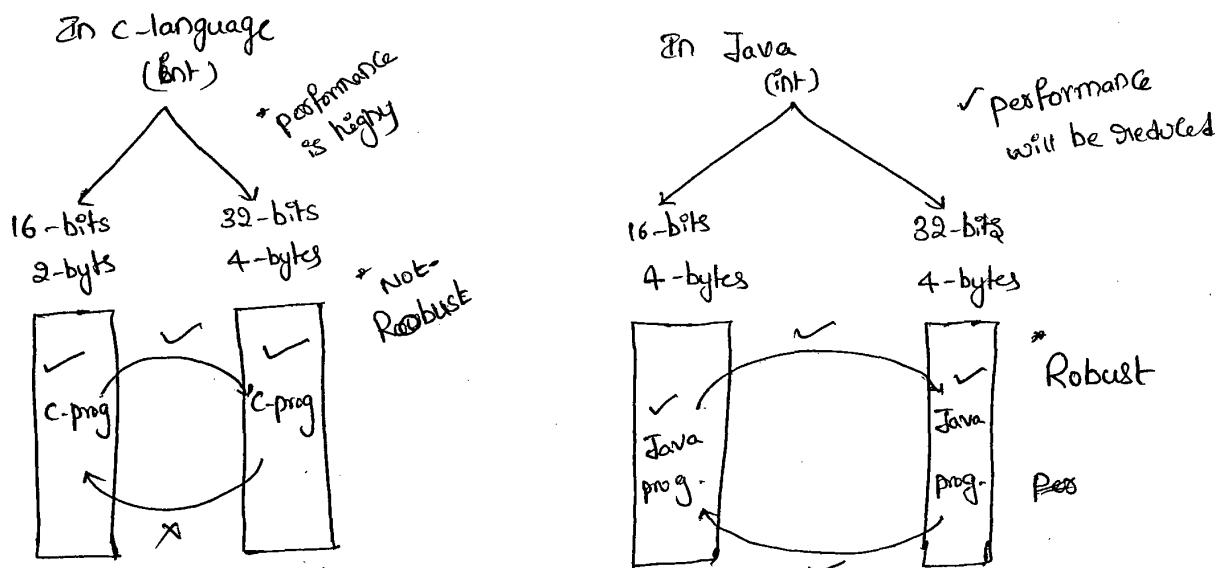
Range : -2^{31} to $2^{31}-1$

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

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~~ failing c program is very very high if we are changing platform. Hence C-language is not considered as Robust.

- But in Java the size of int is always 4-bytes irrespective of any platform. * The main advantage of this approach is the chance of 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.



3/02/11

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

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

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



- | | |
|---|--|
| <ul style="list-style-type: none"> 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 | <ul style="list-style-type: none"> 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

S.O.println(b); true

Ex :-

int x=0;

if(x)

in Java X

{ S.O.println("Hello"); }

else

{ S.O.println("Hi"); }

C++ ✓

}

C.E:- incompatible types

found : int

required : boolean

in Java X

while(1)

in C++ ✓

{ S.O.println("Hello"); }

→ 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 " $> 2^{26}$ " & hence 1-byte is not enough to represent all characters Compulsory We should go for 2-bytes.

Size : 2-bytes

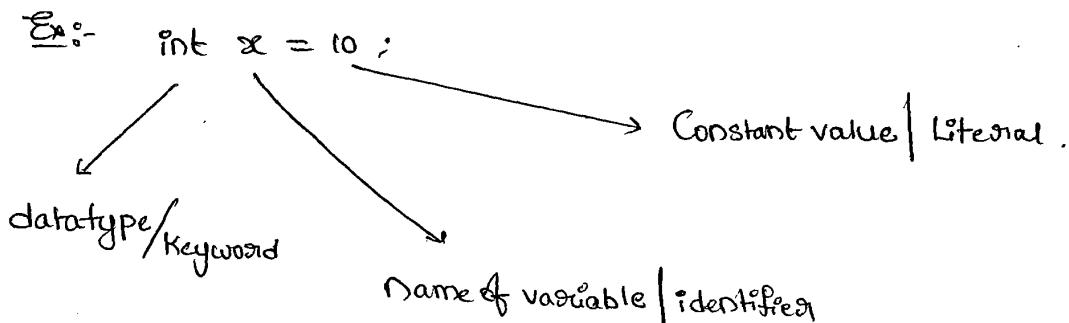
Range : 0 to 65535

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 in C++)

Literals :-

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



Integral Literals :-

→ For the Integral data-types (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

Ex:- `int x = 0x10`

(10)₁₀

`int x = 0X10`

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

Ex:- class Test

{

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

{

`int x = 10;`

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

`int y = 010;`

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

`int z = 0X10;`

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

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

10 8 16

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

}

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 = 0xBEER$` C.E: (after \$) ; 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 with in the range of byte then it treats as byte literal automatically. Similarly short literal also.

Ex:- byte b = 10; ✓

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

Floating point Literals :-

→ Every floating point literal is 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; P.L.P
Found: double
Required: float

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

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

Ex. ✓ double d = 123.4567D;

✗ float f = 123.4567d; C.E:- PLD

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

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

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

✗ 1) float f = 123.456;

✓ 2) double d = 0123.456;

✗ 3) double d = 0x123.456;

✓ 4) double d = 0xfacE; // 64^{20.0}

Because these 3 are not floating point

✓ 5) float f = 0xBear;

So, that values are taking int type.

✓ 6) float f = 0.642; // 418.0

→ 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:- \cancel{X} int $i = 123.456;$ PLP

→ found : double

required : int

✓ double $d = 1.2e3;$

S.O. pln(d); 1200.0

→ we can specify floating point literal even in scientific form

also [exponential form]

Ex:- ✓ 1) double $d = 1.2e3;$

S.O. pln(d); 1200.0

\cancel{X} 2) float $f = 1.2e3;$ C.E.: PLP

→ found : double

✓ 3) float $f = 1.2e3f;$ required : float

O/P:- 1200.0

Boolean Literals:-

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

Q) Which of the following Boolean declarations are valid?

\cancel{X} ① boolean $b = 0;$ C.E.: Incompatible types

→ found : int

\cancel{X} ② boolean $b = \text{True};$ C.E.: Can't find symbol

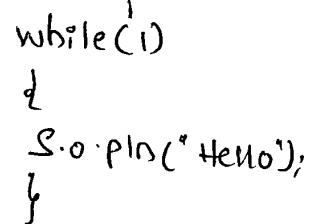
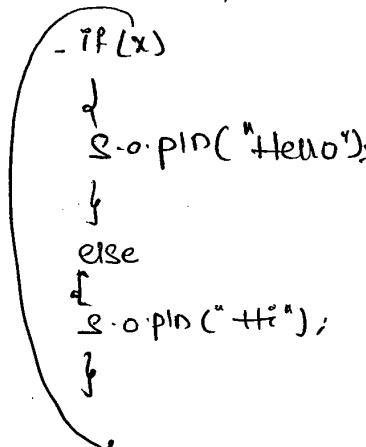
→ required : boolean

✓ ③ boolean $b = \text{true};$ Symbol : variable True

\cancel{X} ④ boolean $b = "true";$ C.E.: Incompatible types

→ found : java.lang.String required : boolean,

Q) Ex:- `int x=0;`



C.E!:- Incompatible types

Found : int

Required : boolean

Ex@:-

`int x=10;` X

```

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 = OXFACE;

✓ 5) char ch = 0640;

3) A char literal can be represented in Unicode representation which is nothing but $\boxed{\text{\uxxxx}}$ 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.

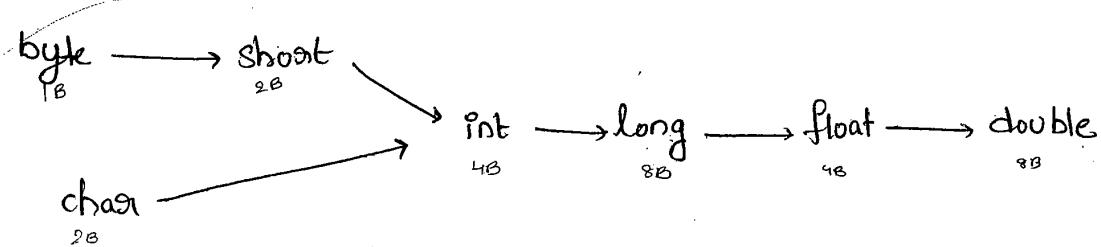
- ✓ 1) char ch = 0xbeaf;
- ✗ 2) char ch = \vbeaf; because ' '
- ✗ 3) char ch = -10;
- ✗ 4) char ch = '*';
- ✓ 5) char ch = 'a';

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 Computer.



Arrays

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 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.

e.g. - X) 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) \checkmark `int[] a,b;` $a \rightarrow 1$
 $b \rightarrow 1$
- 2) \checkmark `int[] a[],b;` $a \rightarrow 2$
 $b \rightarrow 1$
- 3) \checkmark `int[] []a,b;` $a \rightarrow 2$
 $b \rightarrow 2$
- 4) \checkmark `int[] []a,b[];` $a \rightarrow 2$
 $b \rightarrow 3$
- 5) \times `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.

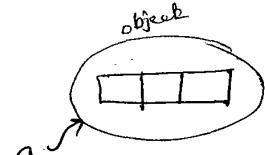
Ex:- `int[] []a, []b;`

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. but 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[];` ~~✗~~ C.E!

`int[] a = new int[3];` ✓

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

Ex:- `int[] a = new int[0];` ✓

→ If we are specifying array size as -ve int value, we will get Runtime Exception saying ~~→~~ NegativeArraySizeException.

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

→ To Specify array size The allowed datatypes are byte, short, int, char. If we are using any other type we will get C.E.

Ex: ① `int[] a = new int['a'];`

$a=97$
 $A=65$

② `byte b = 10;`

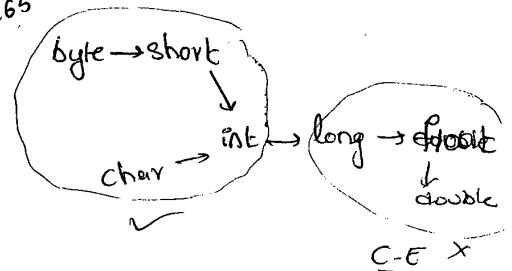
✓ `int[] a = new int[b];`

③ `short s = 20;`

✓ `int[] a = new int(s);`

✗ `int[] a = new int[10L];`

✗ `int[] a = new int[10.5];`



Note:-

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

⇒ Creation of 2D-Arrays:-

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

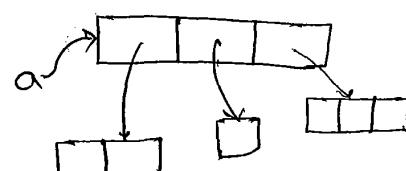
→ The main advantage of this approach is memory utilization will be improved.

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

`a[0] = new int[8];`

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

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



Note:-

In C++, as

Ex 9:

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

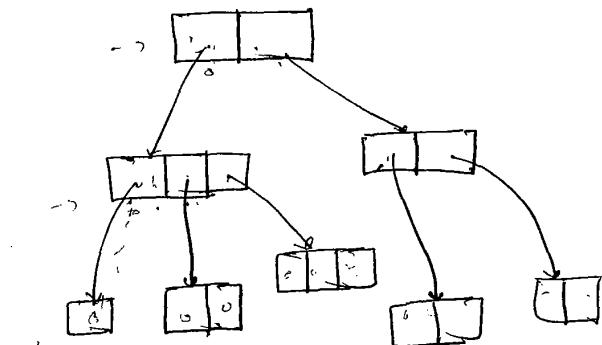
`a[0] = new int[3][];`

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

`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[];`

→ Valid

✓ ② `int[][] a = new int[3][2];`

→ Valid

✓ ③ `int[] a = new int[3][];`

X ④ `int[][] a = new int[][],`

✓ ⑤ `int[][][] a = new int[3][4][5];`

✓ ⑥ `int[][][] a = new int[3][4][],`

X ⑦ `int[][][] a = new int[3][][5];`

→ Valid

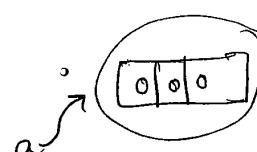
Array Initialization :-

→ 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`

`System.out.println(a[0]); 0`

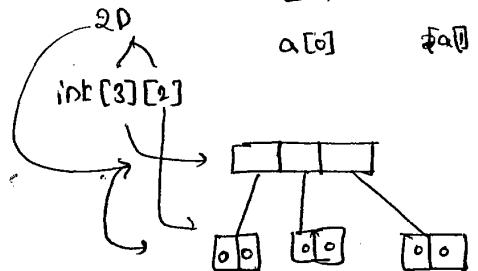
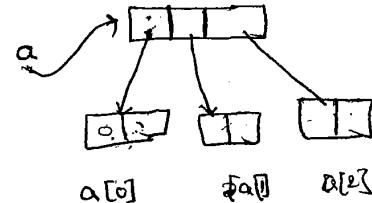


Note:- whenever we are trying to print any object reference internally `toString()` will be called which is implemented as follows.

classname @ hexadecimal_string_of_hashCode.

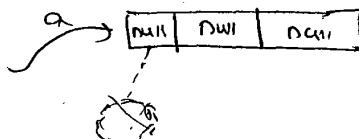
Ex(2) :-

```
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) :-

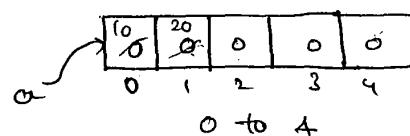
```
int[][] a = new int[3][];
System.out.println(a); [[I@-----
System.out.println(a[0]); null
System.out.println(a[0][0]); R.E. NPE
```



→ 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[3] = 40;
a[50] = 50; → R.E: AIOBE
a[-50] = 60; → R.E: AIOBE
a[10.5] = 30;
```



Note:- C-E :- PEP, 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[3];
a[0] = 10;
a[1] = 20;
a[2] = 30;
a[3] = 40;
    }   → int[] a = {10, 20, 30, 40};
        char
    
```

Ex(2) :- char[] ch = {'a', 'e', 'i', 'o', 'u'};

String[] s = {"Sneha", "Ravi", "Laxmi", "Sunder"};

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

Ex(3) :-

```

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

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

Ex :-

```

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

Ex: `int [][] [] a = {{ {10, 20, 30}, {40, 50}, {60} }, {{70, 80}, {90, 100}, {110}}};` 16

`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

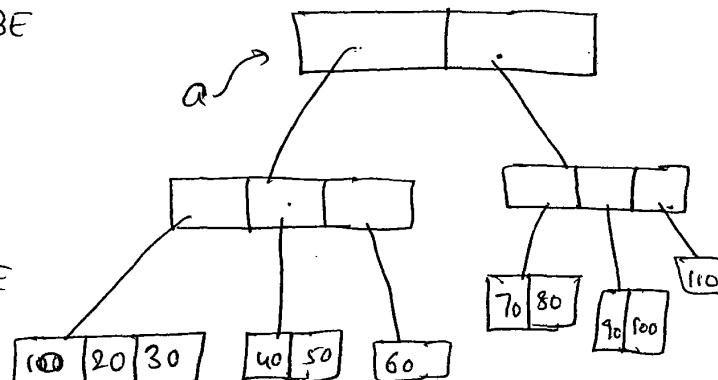
`S.o.println(a[3][1][2]);`; RE:- ALOBE

`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 = {10, 20, 30};` :-

✓ `int x;`

✓ `x=10`

✓ `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 = "doge";`

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

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

↳ C.E.: Cannot find Symbol

Symbol: variable length

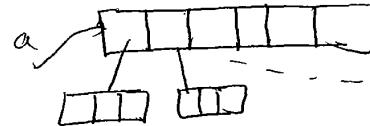
location: java.lang.String.

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

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

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

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



Notes:-

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

Anonymous Array :-

- Sometimes we can create an array with out 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.

Ex:- ~~`New int[n]{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);
    }
}

```

→ 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"};
 ↗ System.out(s[0]); A
 ↗ System.out(s[i]); B
 ↗ System.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] = "10";` C.E! - PLP

found: string
required: int

✗ `a[5] = 10.5;` C.E! - PLP, found: double

required: int

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

byte → short

char → int → long → float → double

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("doege"); → C.E:- Incompatible types

found: String

Required: Number

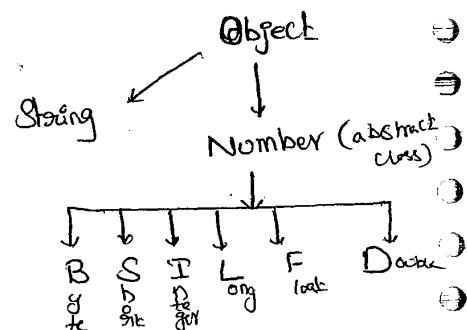
② Object[] a = new Object[10];

✓ a[0] = new Object();

✓ a[1] = new Integer(10);

✓ a[2] = new Double(10.5);

✓ a[3] = new String("doege");



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 its implementation class Objects.

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

arr[0] = new Thread();

X arr[1] = new String("doga"); 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 its child class Objects
3. abstract class type arrays	Its child class objects are allowed.
4. Interface type arrays	its 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 ~~into~~ type. But
char array (char[]) can't be promoted to int[] type.

① int[] a = {10, 20, 30, 40};

char[] ch = {'a', 'b', 'c'};

✓ int[] b = a;

✗ int[] c = ch; C.E! - 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[]

e.g.: Child-type array, we can assign to the parent-type variable.

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

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

✓ Object() a = s;

Ques(2) :-

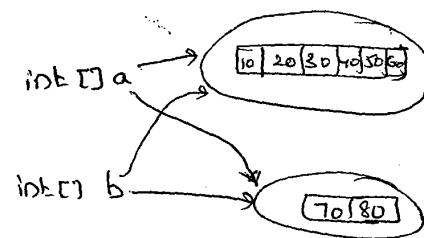
→ When ever we are assigning one array to another array only reference variables will be reassigned but not underlying elements.
Hence types must be matched but not sizes.

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

int [] b = {70, 80};

✓ a = b;

✓ b = a;



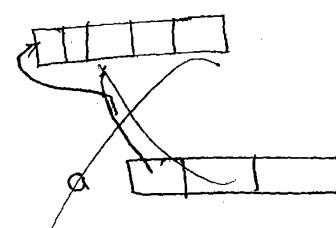
Eg(3): int [][] a = new int [3][2];

a[0] = new int [5];

a[1] = new int [4];

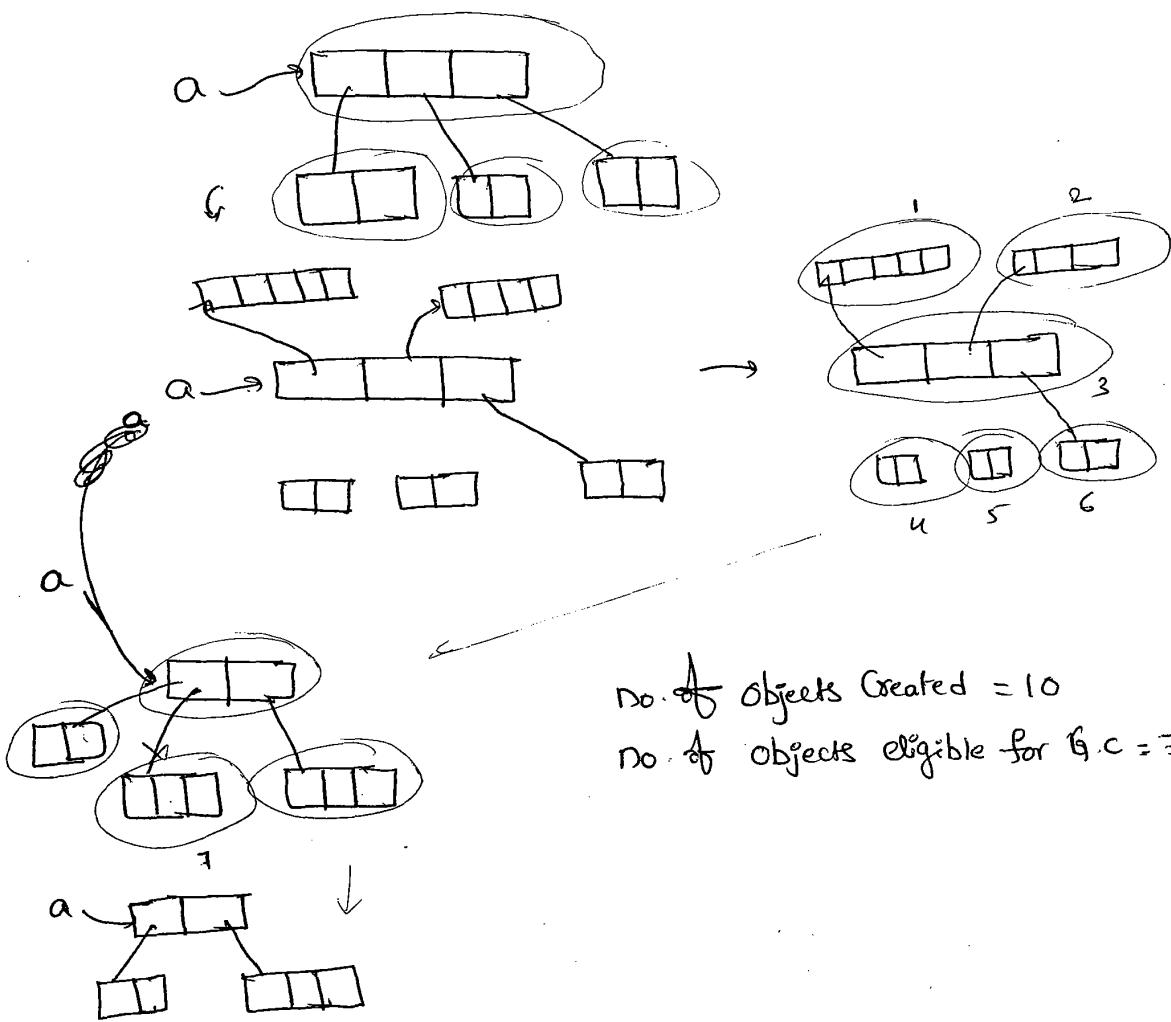
a = new int [2][3];

a[0] = new int [2];



No. of objects Created = 10

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



Case 3:-

→ When ever 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 Compile time Error

e.g.- `int[][] a = new int[3][];`

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

`a[0] = new int[3][2];` → C.E : incompatible types

`a[0] = 0;`

→ found : int[2]()
Required : int[3]

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

Q2

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

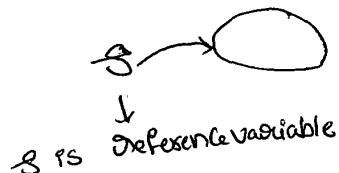
→ Can be used to represent primitive values

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 Block or Constructor.

→ Instance variables Cannot be accessed from insta Static area directly we can access by using object reference.

→ Best 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();

s.out(t.x); so →

} public void m()

{ s.out(x); ↗ so

}

→ 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();

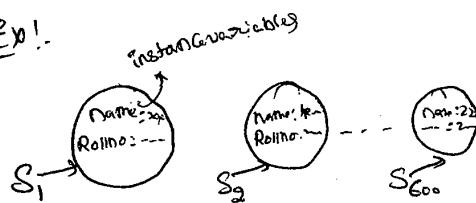
s.out(t.s); null

s.out(t.x); 0

s.out(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;

,

,

,

}

S₁

S₂

S₆₀₀

College-name: durgaSw

→ 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.

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 declared 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.

→ Within 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.pn(Test.x); ✓ 10

S.o.pn(x); ✓ 10

✓ Test t = new Test();

↙ S.o.pn(t.x); ✓ 10

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

→ Eg:- Class Test
 {
 Static int x=10;
 P.s.v.m (String[] args)
 {
 S.o.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); 0
 }
 }

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

* Ex:-

```
class Test
{
    int x=10;
    static int y=20;
    P.S.V.M (String[] args)
}
```

Test t₁=new Test();

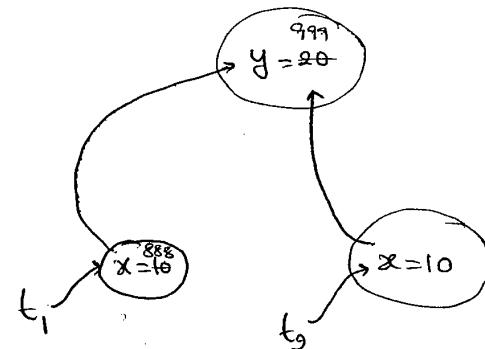
t₁.x=888;

t₁.y=999;

Test t₂=new Test();

S.O.Pln(t₂.x + "----" + t₂.y);

}



t₁.x = 888

t₂.x = 10

t₁.y = 999

t₂.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 their.
- But, if we are performing any change to the static variable, these changes will be reflected for all objects because we are maintaining a single copy.

(iii) Local variables :-

- To meet 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.pn (i + " --- " + d);  
    }  
}
```

* C.E :-

Can't find Symbol
Symbol : variable j
Location: Class Test

→ For the Local Variables JVM won't provide Any default Value,
Compulsory we should perform initialization Explicitly, before Using
That Variable.

Eg:- ①

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

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

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

Eg 3:- Class Test

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

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

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

O/P:- Java Test ←

20

Java Test * 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[] a;

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

{

Test t, = new Test();

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

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

}

Instance level:-

int[] a;

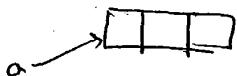
i.e. a=null

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) reference is created but its not initialized (i.e object is not) created. So jvm provides null value to the variable a.

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

Local Level:-

int[] a;

S.o.p(a) { C.E! - variable a might not have 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.

8/03/10 ① 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 arguments Compulsory we should declare a new method. This approach increases length of the code & reduces readability.

→ To resolve these problems Sun people introduced Var-arg method 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(1):-

Class Test

```
p.s. void m1(int... i)
{
    s.o.println("Var-arg method");
}
p.s. v.m(String[] args)
{
    m1();
    m2(10);
    m3(10, 20);
    m4(10, 20, 30, 40);
}
```

Op! :- Var-arg method
 Var-arg method
 " "
 " "

→ Internally Var-arg method is implemented by using single dimensional 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);
}
public static void main(String[] args)
{
    Sum();
    Sum(10, 20);
    Sum(10, 20, 30);
    Sum(10, 20, 30, 40);
}
```

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.println("General method");

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

↳ S.o.println("Var-arg");

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

↳ m1(); var-arg

↳ m1(10); General (only)

↳ m1(10, 20); var-arg

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

Case 6:-

Ex:- Class Test

```
d  
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.

Vari-arg Vs Single dimensional arrays:-

Case(1):-

→ Whenever Single dimensional array present we can replace with vari-arg parameter.

→ $m_1(\text{int[]} x) \Rightarrow m_1(\text{int... } x)$ ✓

$\text{main}(\text{String[]} args) \Rightarrow \text{main}(\text{String... } x)$ ✓

Case(2):-

→ whenever vari-arg parameter present we can't replace with Single dimensional array.

~~$\times \quad m_1(\text{int... } x) \Rightarrow m_1(\text{int[]} x)$~~

29/3/11

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 NoSuchMethodException: main.

Ex:- Class Test
 ↓
 ↓

compile Javac Test.java ✓

run x Java Test → R.E:- NoSuchMethodException: 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 Vararg String parameters.
is `String...`

`main (String[] args) ⇒ main (String... args)` ✓

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

(i) `final`

(ii) `Synchronized`

(iii) `Strictfp`

Ex:- class Test

{

`final static Strictfp Synchronized public void main (String... A)`

{

`S. o. pln ("Hello world");`

}

}

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

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

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

Ans: Nowhere, 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 design S/w");
 }

class C extends P.
 {
 }

javac p.java ✓

java p

Op! - ZLU design S/w

java C

Op! ZLU design S/w

Ex 2:-

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

    S. o. p. n (" I Love ");
}

class C extends P
{
    p. s. v. m (String[] args)
}

    S. o. p. n (" 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 method hiding.

→ Overloading concept is applicable for main() but JVM always calls String[] argument method only. The other method we have to call explicitly.

Ex 1:- class Test

```

    p. s. v. m (String[] args)          o/p:- durgaSw.
}

    S. o. p. n (" durgaSw ");
}

p. s. v. m (int[] args)
}

    S. o. p. n (" is good ");
}
```

Q) Instead of main is it possible to configure any other method as main method? 39

A) Yes, But inside JVM we have to configure some changes then it is possible.

Q) Explain about S.o.println()

A)

Class Test

{

 Static String name = "durga";

}

Test.name.length()

↙

↓
It is a
Static variable of
type String present
in Test class

↓
It is a method
present in
String class

It is a
Class-
name

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

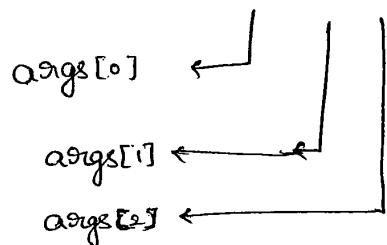
10/10/2011

CommandLine Arguments

CommandLine arguments:

- The arguments which are passing from Command prompt 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



$$\text{args.length} \Rightarrow 3$$

Ex!:- class Test

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

```
    for(int i=0 ; i<args.length ; i++)
    {
```

```
        S.o.println(args[i]);
    }
```

O/P! Java Test ←
for

R.E!:- AIOBE

Java test x y ←

x

y

R.E!:- AIOBE

Ex(1):-

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

Ex:-

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println(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

```
{
    public static void main(String[] args)
    {
        System.out.println(args[0]); Note Book
    }
}
```

Java Test "Note Book"

Ex(2):- class Test

```
{
    public static void main(String[] args)
    {
        String[] args = {"A", "B"};
        args = args;
        for (String s : args)
            System.out.println(s);
    }
}
```

```

Java Test x y ←  

of A  

B  

Java Test x y z ←  

of A  

B  

Java Test ←  

of 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;
}

```

Amazonepet Standard

```

package com.dorgesoft.demo;  

public class Calculator  

{
    public static int Sum(int number1,  

                         int number2)  

    {
        return number1+number2;
    }
}

```

→ Tech-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

2) Coding Standards for Interfaces :-

→ Usually interface names are Adjectives Should Starts with Uppercase Letter & if it Contains multiple words every inner word Should Starts with Uppercase Letter.

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

Note:-

1) 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 Uppercase Letter. (CamelCase).

Ex:-

run()
sleep()
eat()
init()
wait()
join()

} → Verbs

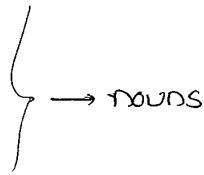
getName()
setSalary()

} Verb + noun

4) Coding Standards for Variables :-

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

Ex! Name
roll no
mobile Number



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 gettor & Setter methods.

Ex:-

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

→ ends with Bean Is not official Conventions 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;
}
  
```

① Coding Standards for Listeners :-

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)

X ② public void registerMyActionListener(MyActionListener l)

X 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)

X ② public void unregisterMyActionListener(MyActionListener l)

X ③ public void deleteMyActionListener(MyActionListener l)

X ④ public void removeMyActionListener(ActionListener l)

Note:-

In Java bean Coding Standards & Listener Concept 1 compulsory.

卷之三

1

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 Operator 12

Conditional Operator 13

New Operator 13

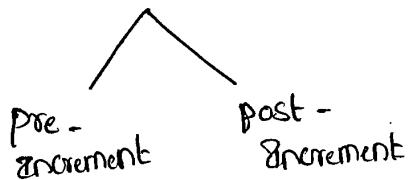
[] operator 13

Operator precedence 14

Evaluation Order of Java operands. 14

Increment & Decrement Operators:

Increment



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

Decrement



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

Expression

Initial value

of x

Final value

of x

Final value

of y

`y = ++x;`

4

5

5

`y = x++;`

4

5

4

`y = --x;`

4

3

3

`y = x--;`

4

3

4

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

`int x = 4;`

~~X~~ `int y = ++4;` C.E: unexpected type

`S.output(y);`

↳ found : Value ②
 ↳ required : Variable ①

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

`int x = 4;`

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

~~`S.o.p(y);`~~

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

→ after score - it 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 = 0.5;`
~~d++;~~
`S.o.p(d); // 11.5`

② `char ch = 'a';`
~~ch++;~~
`S.o.p(ch); // b`

③ `boolean b = true;`

~~`++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 = (byte)(b+1);

`b++ ;` \Rightarrow `b = (\text{typedef } 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.0 pln (10+0.0); // 10.0

`int + long = long`

Sopln('a'+'b') ; 195

long + float = float

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

double + char = double

$$c_{\text{ba}g_1} + c_{\text{ba}g_2} = \text{opt}$$

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 ArithmeticException. (AE = 1 by zero)

Eq:-

S. o. pln (10/0); R.E.: A.E.: 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

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

Negative-Infinity = -infinity

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

Eg:- ①. S.o.pln(10/0.0) ; Infinity

②. S.o.pln(-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.pln(0/0.0) ; Nan.

* $\text{S.o.p}(0.0/0); \text{NaN}$

* $\text{S.o.p}(-0/0.0); \text{NaN}$

40

Ex: * $\text{public static void Sqrt(double d);}$

$\text{S.o.println(math.Sqrt(4)); /2.0}$

$\text{S.o.println(math.Sqrt(-4)); NaN.}$

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

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

at $x=10$

$\text{S.o.p}(10 > \text{float.NaN}); \text{false}$

$\text{S.o.p}(10 < \text{float.NaN}); \text{false}$

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

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

$\text{S.o.p}(\text{float.NaN} == \text{float.NaN}); \text{false}$

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

$x > \text{NaN}$
 $x \geq \text{NaN}$
 $x < \text{NaN}$
 $x \leq \text{NaN}$
 $x == \text{NaN}$

Conclusion about A.E (Arithmetic 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 & sometimes acts as String arithmetic Operator. (or) String Concatenation Operator.

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

String d = "Shanthi";

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

S.o.p(a+b+d+c); 30Shanthi30

S.o.p(d+a+b+c); Shanthi102030

S.o.p(a+d+b+c); 10Shanthi2030.

$d+a+b+c$
Shanthi10+20+30
Shanthi1020+30
Shanthi102030

→ 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 = "Shanthi";

✗ a = b+c; ^{total String} C.E:- Incompatible type; found : String
Required : int

✓ c = a+c; ^{total String}

✓ b = a+b;
^{int} _{int}

✗ 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, boolean

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

Eg:- 1) "Shanthi" < "Shanthi" ✗

2) "durga" < "durga123" ✗

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 < 20 < 30)

boolean

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



Eg:- String s₁ = new String("durga");

String s₂ = new String("durga");

s₁ → durga

S.o.println(s₁ == s₂); false (reference)

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

s₂ → durga

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

→ These are $==$, $!=$

* we can apply Equality operators for Every primitive type including

boolean types.

o/p

Eg:-	
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

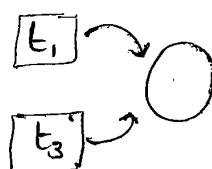
Ex(i): Thread $t_1 = \text{new Thread}();$

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

Thread $t_3 = t_1;$

$\times \text{S.o.p}(t_1 == t_2);$ False

$\checkmark \text{S.o.p}(t_1 == t_3);$ True



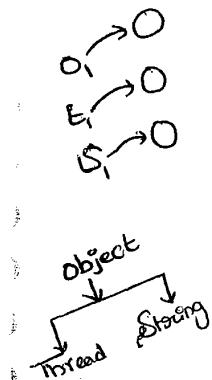
* To apply Equality Operators b/w the object references Compulsory

These 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].

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



```
Thread t1 = new Thread();
```

String s_i = New String ("shant"),

S.o.p($t_1 = s_1$); CE :- Incompatible types Thread & JavaLang

`S.o.p(ti == 0);` F `java.lang.String`

$$S \circ p(S_1 = \emptyset), \quad F$$

→ for any object reference s_1 , if s_1 is pointing to any object

$\text{g1} == \text{null}$ is always, false, otherwise g1 Contains null value

) → So, $null == null$ is always True.

Q) Note:-

* In General, operator overloading for difference Composition

where as.equals() method ment for Content Comparison.

InstanceOf Operator

(instanceof) ✓

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

SyD:-

9 instanceof x

any preference type

class / interface.

instanceof
Hashtree
Strictfp

Ex:- short $S = 15;$

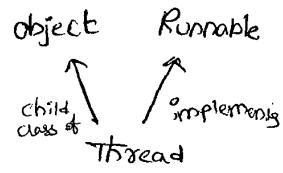
Boolean b;

$$b = (\text{s instance of Short})$$

b = (5 instanceof Number)

~~Ego~~ i) 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 : Strong

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

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

- ✓ S.o.P (0 instanceOf String); false

→ for any class ~~or~~ interface of X, null instance of X always returns "false".

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

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

Object 0 = iter.next(1)

if (0 instanceof Student)

else if(0 instances of C_2)

Apply customer details

Bit-wise Operators :-

- (1) & → AND → if Both operands ^(of arguments) are True then Result is True
 (2) | → OR → if atleast 1 operand is T " " T
 (3) ^ → X-OR → if Both operands are different " " T

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

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

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

Ex(1):- 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 operators 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) :-

(Filed)

S.o.println($\sim T$); CE: operator \sim can't be applied to boolean.

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

Ex:- i) S.o.println($\sim \text{True}$);

CE: operator \sim can't be applied to boolean.

✓ ii) S.o.println(~ 4); -5

$$4 \equiv 0000\ 0000 \quad \dots \quad 0100$$
$$\sim 4 = \boxed{1} \ 1111 \ 1111 \quad \dots \quad 1011$$

\downarrow
2's Complement

0 → +ve
1 → -ve

-ve

One's Comp

$$\begin{array}{r} 000\ 0000 \quad \dots \quad 0100 \\ 111\ 1111 \quad \dots \quad 1011 \\ \hline 000 \quad \dots \quad 0101 \end{array}$$

add '1' to 1's Comp
is 2's Comp

2's Comp

-ve 5

$\therefore -5$

Note:

- The most significant bit represents Sign bit. 0 means +ve no, 1 means -ve no.
- +ve no. will be represented directly in the memory. whereas as -ve no's will be represented in 2's Complement form.

Boolean Complement Operator (!) :-

→ 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.

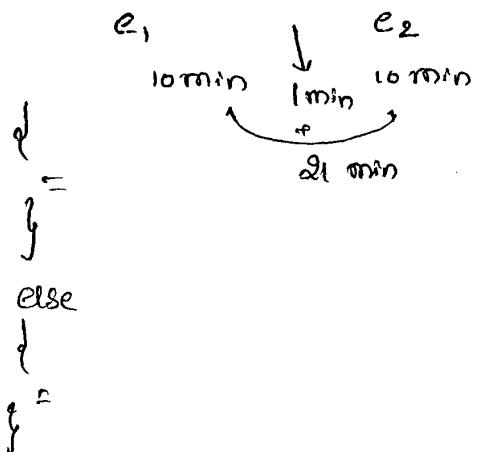
! ⇒ we can apply only for boolean types but not for integral types.

** 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.

<code>&, </code>	<code>&&, </code>
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.

Sol:- `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

{

$++y;$

}

So. $\text{printf}(x + "----" + y);$

Ques:-

	x	y
$\&$	11	17
1	12	16
11	12	15
$\&\&$	11	17

```

⑧ 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 b, that is R.E.

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

TypeCast Operators :-

→ There are 2 types of primitive type Castings.

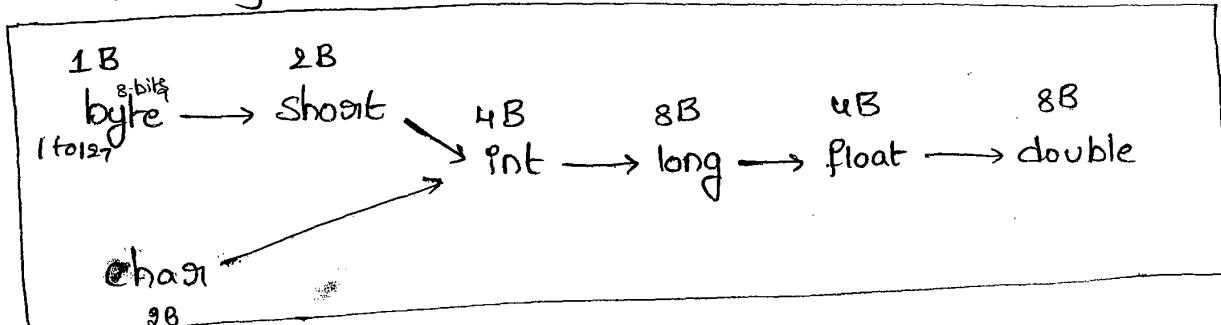
1. Implicit type Casting
2. Explicit type Casting.

Implicit Type Casting :-

- 1) Compiler is responsible to perform this type casting
- 2) This Typecasting is required whenever 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

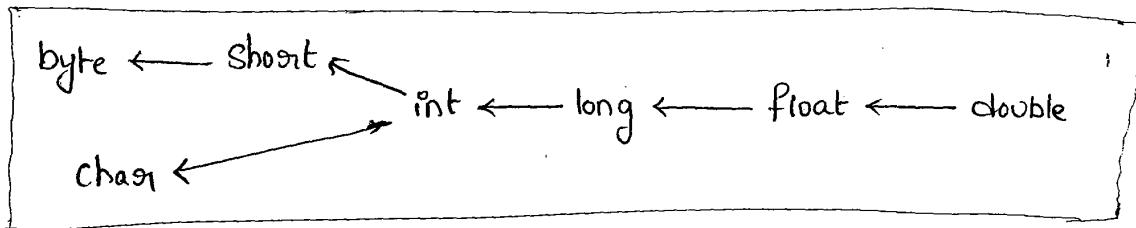


Ex(1) :-

- 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`
- ③ `A=97, B=98, 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 TypeCasting.
- 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

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

① \times byte $b = 130$;

\checkmark byte $b = (\text{byte}) 130$;

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

(32-bit)

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

Q7

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

-ve

\downarrow 2's Complement

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

$$\begin{array}{r} 1111101 \\ ,1 \\ \hline 1111110 \end{array}$$

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

Ans: 126

$$\therefore -126$$

②

int $i = 150$;

short $s = (\text{short}) i$;

$s.\text{o.println}(s) \neq 150$

$$150 \equiv 0000\ldots \underline{\underline{010010110}} \quad 32 \text{ bits}$$

short $s \equiv 0000\ldots \underline{\underline{010010110}} \rightarrow 2 \text{ Bytes} = \text{short} = 16 \text{-bits}$

+ve

\downarrow don't apply 2's Comp.

$$\therefore s = 150$$

③ int $x = 150$;

byte $b = (\text{byte}) x$;

short $s = (\text{short}) x$;

$s.\text{o.println}(b); -106$

$s.\text{o.println}(x); 150$

$$150 \equiv 0000\ldots \underline{\underline{010010110}}$$

$$\text{byte } b = \underline{\underline{010010110}}$$

-ve

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

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

$$\begin{array}{r} 1101001 \\ ,1 \\ \hline 1101010 \end{array}$$

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

10/2/11

- whenever we are assigning floating point datatype values to the integral data types by Explicit type Casting the digits after the decimal point will be lost.

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 1248

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 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 operator to form Compound assignment operator.

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

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

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

(14)

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

$\text{Ex } \textcircled{1}$ byte b = 10; b = b + 1; S.o.println(b); <u>C-E:</u> 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 <hr/> byte b = 127; b += 3; S.o.println(b); -126
--	---	---

Ex $\textcircled{2}$:
~~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~~ →

~~a > b is T then 40~~

~~a > b is F then 50~~

~~S.o.println(x); 50~~

$a+b$ → binary operator

$++a$ → unary "

$(a+n)? a : b$ → ternary "

→ Nesting of Conditional operator is possible.

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

`int x = (a>50) ? 777 : ((b>100) ? 888 : 999);`

F F

`S.o.println(x); 999`

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

✓ | `byte c = (true) ? 40 : 50;` ✓ $a < 12 \rightarrow T$
 ✓ | `byte c = (false) ? 40 : 50;` ✗ $a < b \times C.E$
 | don't compare these variables

✗ | `byte c = (a < b) ? 40 : 50;` C.E.: PLP
 ✗ | `byte c = (a > b) ? 40 : 50;` found: int
 | required: byte.

→ `final int a=10, b=20;`

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

New Operator:-

→ We can use this operator for creation of objects.

→ In Java there is no Delete operator. because destruction of

useless object is responsibility of Garbage Collector.

[] Operator:-

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

Operator precedence :-

1. Unary operators :-

[] , $x++$, $x--$
 $++x$, $--x$, \sim , !

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 :-

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

Evaluation Order of operands :-

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

Ex:- class EvaluationOrderDemo

```
p.s.v.m (String [] args)
{
    S.o.p (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;

S.o.println(x); //

}

1st increment

2nd place init into x

int x = 10;

x = x++;

S.o.println(x); 10

1st place x = 10

∴ x = 10++

∴ x = 11

but last operation is

x = 10

Ex(3) :-

④

int x = 0;

(+2)⁸

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

S.o.p(x); 8

x = 0 x x x 4

x++ = 1

x++ = 2

3

4

Ex 4:-

int x = 0;

x += ++x + x++;

S.o.println(x); 2

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

= 0 + 1 + 1

x = 2

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
7010
7011
7012
7013
7014
7015
7016
7017
7018
7019
7020
7021
7022
7023
7024
7025
7026
7027
7028
7029
7030
7031
7032
7033
7034
7035
7036
7037
7038
7039
70310
70311
70312
70313
70314
70315
70316
70317
70318
70319
70320
70321
70322
70323
70324
70325
70326
70327
70328
70329
70330
70331
70332
70333
70334
70335
70336
70337
70338
70339
70340
70341
70342
70343
70344
70345
70346
70347
70348
70349
70350
70351
70352
70353
70354
70355
70356
70357
70358
70359
70360
70361
70362
70363
70364
70365
70366
70367
70368
70369
70370
70371
70372
70373
70374
70375
70376
70377
70378
70379
70380
70381
70382
70383
70384
70385
70386
70387
70388
70389
70390
70391
70392
70393
70394
70395
70396
70397
70398
70399
703100
703101
703102
703103
703104
703105
703106
703107
703108
703109
703110
703111
703112
703113
703114
703115
703116
703117
703118
703119
7031100
7031101
7031102
7031103
7031104
7031105
7031106
7031107
7031108
7031109
7031110
7031111
7031112
7031113
7031114
7031115
7031116
7031117
7031118
7031119
70311100
70311101
70311102
70311103
70311104
70311105
70311106
70311107
70311108
70311109
70311110
70311111
70311112
70311113
70311114
70311115
70311116
70311117
70311118
70311119
703111100
703111101
703111102
703111103
703111104
703111105
703111106
703111107
703111108
703111109
703111110
703111111
703111112
703111113
703111114
703111115
703111116
703111117
703111118
703111119
7031111100
7031111101
7031111102
7031111103
7031111104
7031111105
7031111106
7031111107
7031111108
7031111109
7031111110
7031111111
7031111112
7031111113
7031111114
7031111115
7031111116
7031111117
7031111118
7031111119
70311111100
70311111101
70311111102
70311111103
70311111104
70311111105
70311111106
70311111107
70311111108
70311111109
70311111110
70311111111
70311111112
70311111113
70311111114
70311111115
70311111116
70311111117
70311111118
70311111119
703111111100
703111111101
703111111102
703111111103
703111111104
703111111105
703111111106
703111111107
703111111108
703111111109
703111111110
703111111111
703111111112
703111111113
703111111114
703111111115
703111111116
703111111117
703111111118
703111111119
7031111111100
7031111111101
7031111111102
7031111111103
7031111111104
7031111111105
7031111111106
7031111111107
7031111111108
7031111111109
7031111111110
7031111111111
7031111111112
7031111111113
7031111111114
7031111111115
7031111111116
7031111111117
7031111111118
7031111111119
70311111111100
70311111111101
70311111111102
70311111111103
70311111111104
70311111111105
70311111111106
70311111111107
70311111111108
70311111111109
70311111111110
70311111111111
70311111111112
70311111111113
70311111111114
70311111111115
70311111111116
70311111111117
70311111111118
70311111111119
703111111111100
703111111111101
703111111111102
703111111111103
703111111111104
703111111111105
703111111111106
703111111111107
703111111111108
703111111111109
703111111111110
703111111111111
703111111111112
703111111111113
703111111111114
703111111111115
703111111111116
703111111111117
703111111111118
703111111111119
7031111111111100
7031111111111101
7031111111111102
7031111111111103
7031111111111104
7031111111111105
7031111111111106
7031111111111107
7031111111111108
7031111111111109
7031111111111110
7031111111111111
7031111111111112
7031111111111113
7031111111111114
7031111111111115
7031111111111116
7031111111111117
7031111111111118
7031111111111119
70311111111111100
70311111111111101
70311111111111102
70311111111111103
70311111111111104
70311111111111105
70311111111111106
70311111111111107
70311111111111108
70311111111111109
70311111111111110
70311111111111111
70311111111111112
70311111111111113
70311111111111114
70311111111111115
70311111111111116
70311111111111117
70311111111111118
70311111111111119
703111111111111100
703111111111111101
703111111111111102
703111111111111103
703111111111111104
703111111111111105
703111111111111106
703111111111111107
703111111111111108
703111111111111109
703111111111111110
703111111111111111
703111111111111112
703111111111111113
703111111111111114
703111111111111115
703111111111111116
703111111111111117
703111111111111118
703111111111111119
7031111111111111100
7031111111111111101
7031111111111111102
7031111111111111103
7031111111111111104
7031111111111111105
7031111111111111106
7031111111111111107
7031111111111111108
7031111111111111109
7031111111111111110
7031111111111111111
7031111111111111112
7031111111111111113
7031111111111111114
7031111111111111115
7031111111111111116
7031111111111111117
7031111111111111118
7031111111111111119
70311111111111111100
70311111111111111101
70311111111111111102
70311111111111111103
70311111111111111104
70311111111111111105
70311111111111111106
70311111111111111107
70311111111111111108
70311111111111111109
70311111111111111110
70311111111111111111
70311111111111111112
70311111111111111113
70311111111111111114
70311111111111111115
70311111111111111116
70311111111111111117
70311111111111111118
70311111111111111119
703111111111111111100
703111111111111111101
703111111111111111102
703111111111111111103
703111111111111111104
703111111111111111105
703111111111111111106
703111111111111111107
703111111111111111108
703111111111111111109
703111111111111111110
703111111111111111111
703111111111111111112
703111111111111111113
703111111111111111114
703111111111111111115
703111111111111111116
703111111111111111117
703111111111111111118
703111111111111111119
7031111111111111111100
7031111111111111111101
7031111111111111111102
7031111111111111111103
7031111111111111111104
7031111111111111111105
7031111111111111111106
7031111111111111111107
7031111111111111111108
7031111111111111111109
7031111111111111111110
7031111111111111111111
7031111111111111111112
7031111111111111111113
7031111111111111111114
7031111111111111111115
7031111111111111111116
7031111111111111111117
7031111111111111111118
7031111111111111111119
70311111111111111111100
70311111111111111111101
70311111111111111111102
70311111111111111111103
70311111111111111111104
70311111111111111111105
70311111111111111111106
70311111111111111111107
70311111111111111111108
70311111111111111111109
70311111111111111111110
70311111111111111111111
70311111111111111111112
70311111111111111111113
70311111111111111111114
70311111111111111111115
70311111111111111111116
70311111111111111111117
70311111111111111111118
70311111111111111111119
703111111111111111111100
703111111111111111111101
703111111111111111111102
703111111111111111111103
703111111111111111111104
703111111111111111111105
703111111111111111111106
703111111111111111111107
703111111111111111111108
703111111111111111111109
703111111111111111111110
703111111111111111111111
703111111111111111111112
703111111111111111111113
703111111111111111111114
703111111111111111111115
703111111111111111111116
703111111111111111111117
703111111111111111111118
703111111111111111111119
7031111111111111111111100
7031111111111111111111101
7031111111111111111111102
7031111111111111111111103
7031111111111111111111104
7031111111111111111111105
7031111111111111111111106
7031111111111111111111107
7031111111111111111111108
7031111111111111111111109
7031111111111111111111110
7031111111111111111111111
7031111111111111111111112
7031111111111111111111113
7031111111111111111111114
7031111111111111111111115
7031111111111111111111116
7031111111111111111111117
7031111111111111111111118
7031111111111111111111119
70311111111111111111111100
70311111111111111111111101
70311111111111111111111102
70

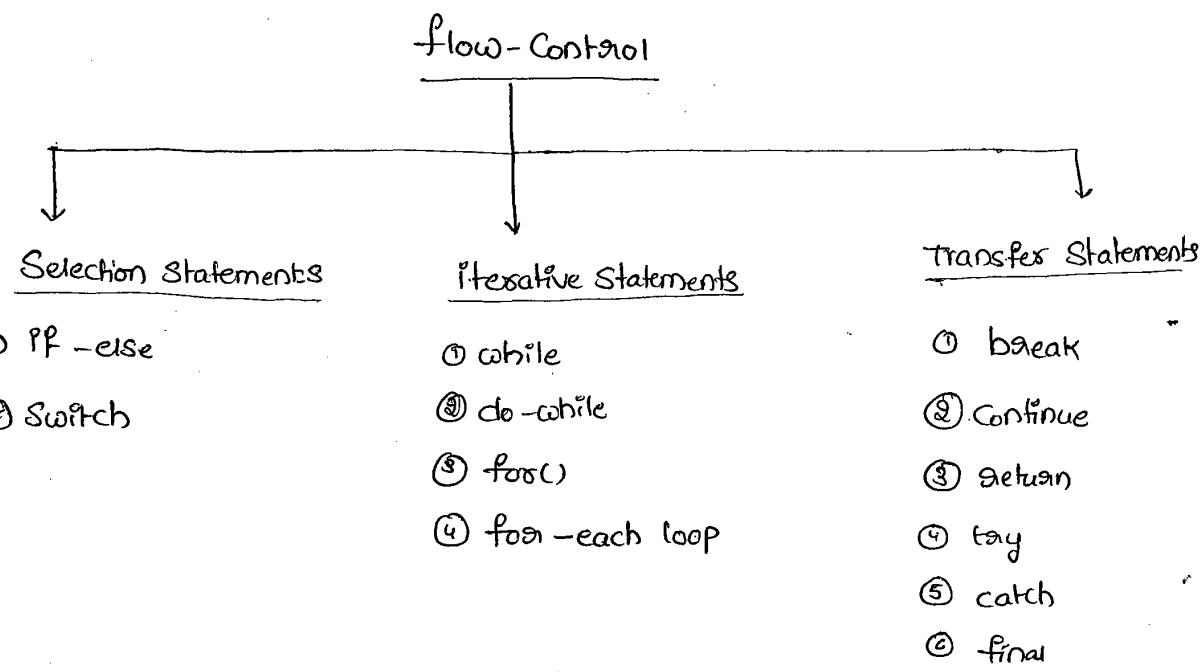


Flow Control

16/05/2011 52

Flow Control :-

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



a) Selection Statements:-

c) 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)
  ↓
  S.o.println("Hello");
}
else
  ↓
  S.o.println("Hi");
}
```

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

② int $x = 10$

```
if ( $x \geq 20$ )
  ↓
  S.o.println("Hello");
}
else
  ↓
  S.o.println("Hi");
}
```

③ int $x = 10;$

```
if ( $x \geq 20$ )
  ↓
  S.o.println("Hello");
}
else
  ↓
  S.o.println("Hi");
}
```

O/P:- Hi ✓

④

boolean $b = \text{false};$

```
if ( $b = \text{true}$ )
  ↓
  S.o.println("Hello");
}
else
  ↓
  S.o.println("Hi");
}
```

O/P:- Hello ✓

⑤

boolean $b = \text{false};$

```
if ( $b == \text{true}$ )
  ↓
  S.o.println("Hello");
}
else
  ↓
  S.o.println("Hi");
}
```

O/P:- Hi ✓

5/3

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

Ex:-

if (true)
S.o.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 never recommended to use if-else, we should go for Switch statement.

Syn:- Switch(x)



Case1 : Action1;

Case2 : Action2;



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

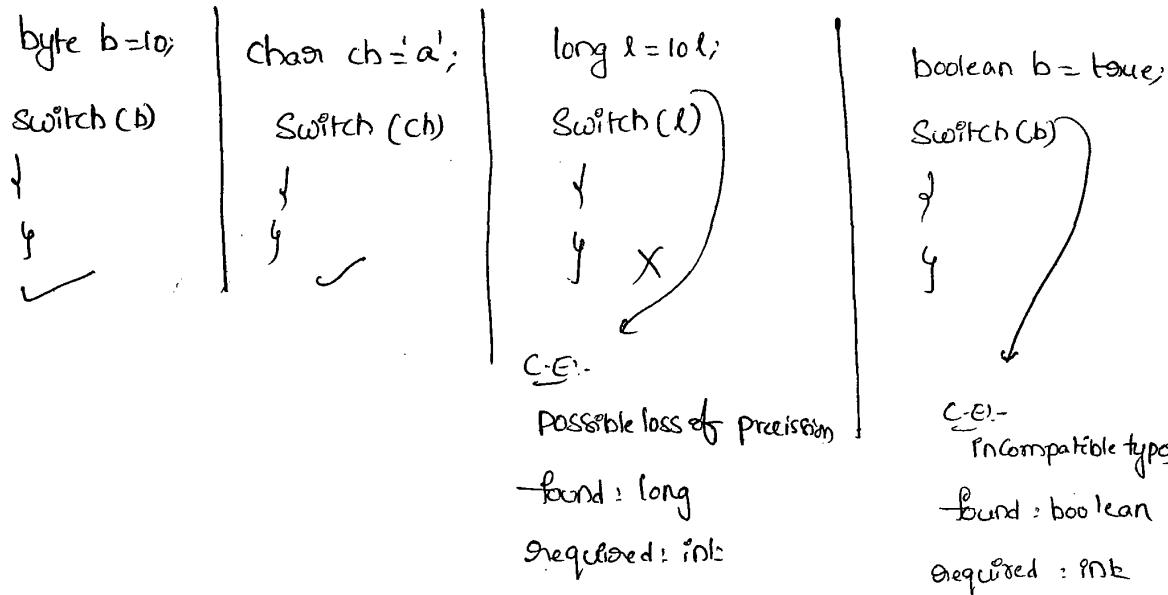
int

char

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

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

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

Ex:-

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

```

) ex:- byte b=10;
)   Switch(b)
)   |
)   Case 10:
)     S.o.println("10");
)   Case 100:
)     S.o.println("100");
)   Case 1000:
)     S.o.println("1000");
)   |
)   C.E:- possible loss of precision
)   found: byte int
)   required: byte.
) 
```

```

byte b=10;
Switch(b+1) → int type
|
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:  
        cout<<"10";  
    case y:  
        cout<<"20";  
}
```

Suppose final int y=20;
Case y:
cout<<"20";

C.E.: Constant Expression required.

→ 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:

cout<<"10";

Case 10+20:

cout<<"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");`

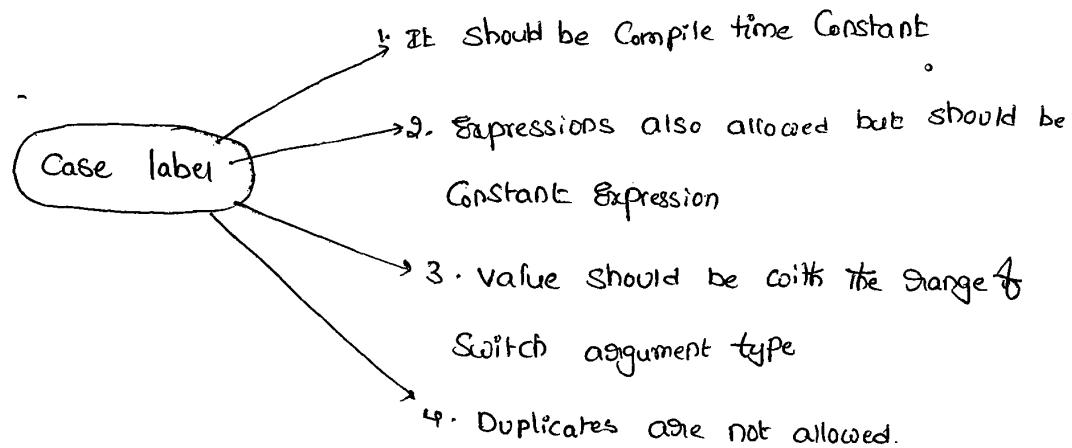
`Case 'a':`

`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.pln("0");

Case 1:

 S.o.pln("1");

 break;

Case 2:

 S.o.pln("2");

default:

 S.o.pln("def");

}

Op:-

if $x=0$:-

 0
 1

if $x=1$:-

 1

if $x \geq 2$

 2
 def

if $x \geq 3$

 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");` $\frac{x=0}{0}$ $\frac{x=1}{2}$

`Case 0:` `S.o.println("0");`

`break;`

`Case 1:` `S.o.println("1");`

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

$\frac{2 \geq 2}{2}$ $\frac{x \geq 3}{\text{def}}$

(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())`
 ↓
 = = = **Result Set**
 {

② `while (itr.hasNext())`
 ↓
 = = = **Iterator**
 {

③ `while (e.hasMoreElements())`
 ↓
 = = = **enumeration**
 {

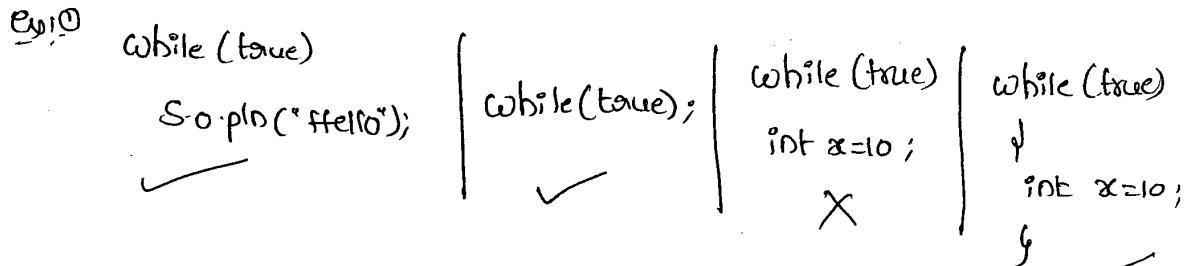
Syntax :-
`while (b)` ↗ boolean type
 ↓
 Action
 {

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

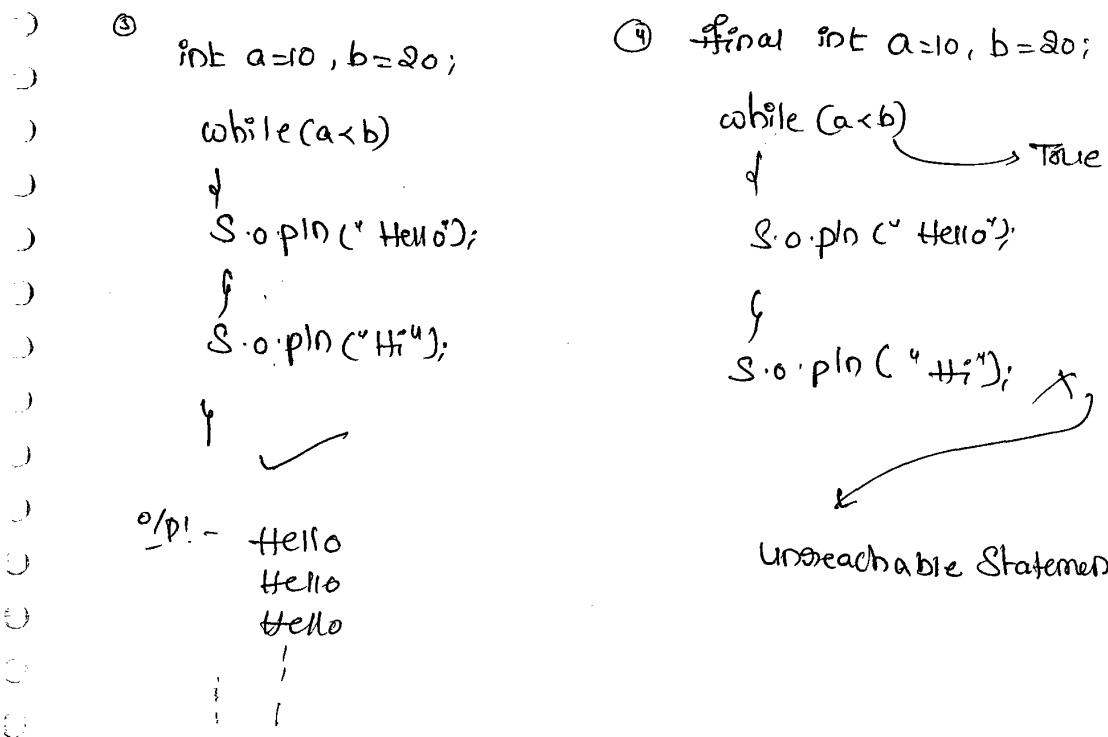
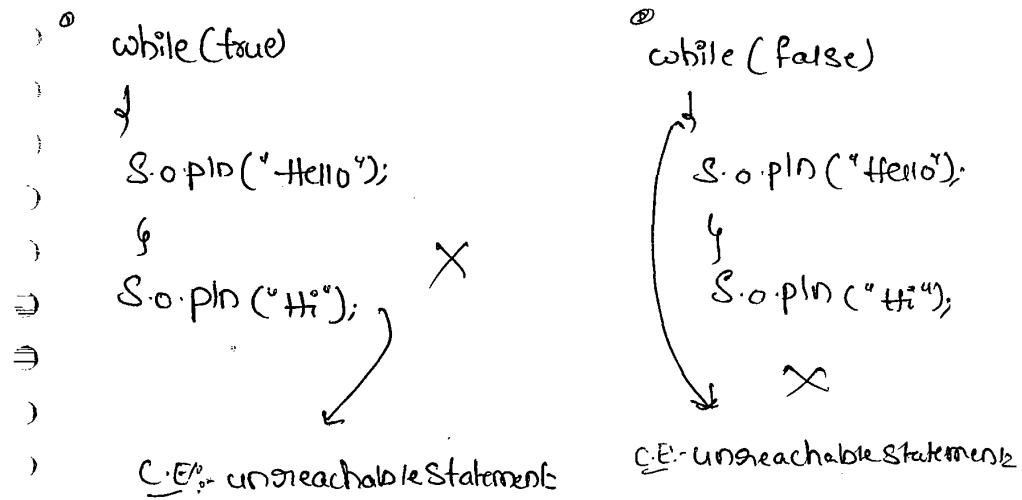
Ex:- `while (1)`
 ↓
 `S.o.println("Hello");`
 {

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(2) :-



④ 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
|
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
  
```

58

③

```

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

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
  
```

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

د

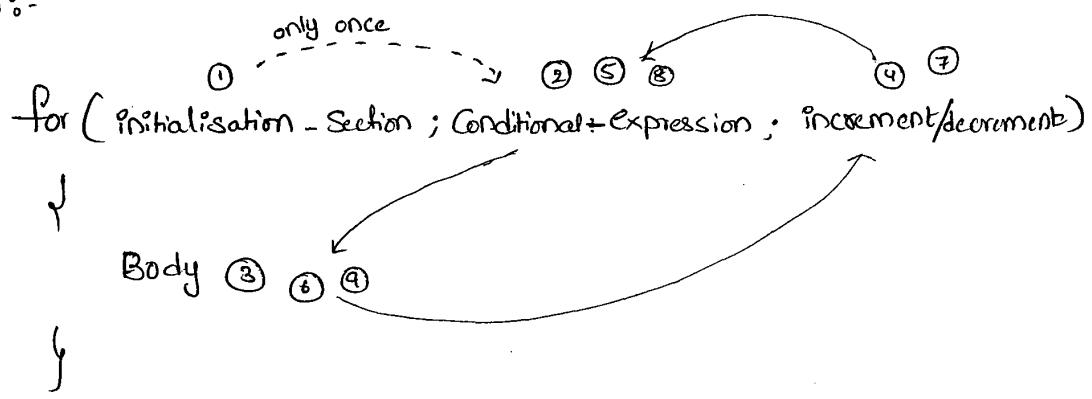
د

د

for() :-

→ This is the most commonly used loop

Syntax:-



- Usually braces are optional & without curly braces we can take
- Only one statement which should not be declarative Statement.

(a) Initialization-Section :-

→ This will be executed only once.

→ Usually we are perf declaring and performing initialization for
the variables in this section.

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

Ex:- ① int i=0, j=0; ✓

② int i=0, byte b=0; X

③ int i=0, int j=0; X

→ 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++)

}

S.o.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".

Encrement & decrement Section :-

→ We can take any valid java statement including S.o.p() also.

Ex:- int i=0;

for(S.o.println("Hello") ; i<3 ; S.o.println("Hi"))

}

Step1(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:- $\text{for}(\text{;}; \text{;})$; \checkmark Statement
& it is True.

⇒ Represent infinite loop

Note:-

;
is a valid Java Statement

Ex:-

<pre> for(int i=0; true; i++) { System.out.println("Hello"); } System.out.println("Hi"); <u>C.E!:- unreachable</u> </pre>	<pre> for(int i=0; false; i++) { System.out.println("Hello"); } System.out.println("Hi"); <u>C.E!:- unreachable</u> </pre>	<pre> for(int i=0; ; i++) { System.out.println("Hello"); } System.out.println("Hi"); <u>C.E!:- unreachable</u> </pre>
<pre> int a=10, b=20; for(int i=0; a<b; i++) { System.out.println("Hello"); } System.out.println("Hi"); <u>O/P:- Hello</u> \checkmark </pre>	<pre> final int a=10, b=20; for(int i=0; a<b; i++) { System.out.println("Hello"); } System.out.println("Hi"); <u>O/P:- C.E!:- unreachable statement.</u> </pre>	

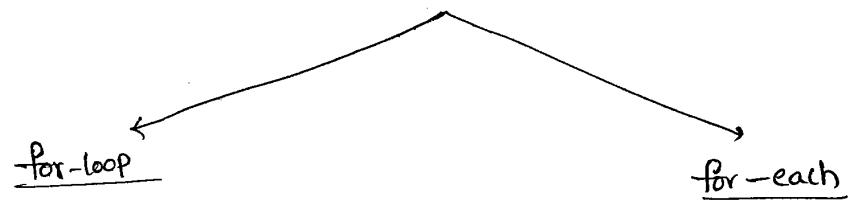
for-each() Loop :- (Enhanced for loop) :-

→ Introduced in Java. 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}};

for (int i=0; i<a.length; i++)
|
for (int j=0; j<a[i].length; j++)
|
System.out.println(a[i][j]);
|
|
|
10
20

for (int[] x: a)
↓
for (int y: x)
↓
System.out.println(y);
|
|
|
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.println(i);
```

```
:
```

Class Test

P. S. V. M (→)

```
int i=10;
```

```
l:
```

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

```
if (i == 10)
```

```
    break l;
```

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

```
}
```

```
System.out.println("End");
```

Output:
Hello
End

→ If we are using break statement anywhere else we will get
Compiletime Error

Ex:- Class Test

```
    }  
    p - S - v - m ( → )  
    }  
    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 $x=10;$
 $\text{if } (x == 10)$
 Continue; → x
 $\text{S.o.p}(“Hello”);$ 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:- $l_1:$
 $\text{for}(\dots)$
 ↓ $l_2:$
 $\text{for}(\dots)$
 ↓
 $\text{for}(\dots)$
 ↓
 break $l_1;$
 break $l_2;$
 break;

Ex 2:- $l_1:$
 $\text{for}(\text{int } i=0; i<3; i++)$
 ↓
 $\text{for}(\text{int } j=0; j<3; j++)$
 ↓
 $\text{if } (i=j)$
 break;
 $\text{S.o.p}(i + \dots + j);$

break:-
 $1 \dots 0$
 $2 \dots 0$
 $2 \dots 1$

break l_1 :-

No output

Continue :- $0 \dots 1 \quad 2 \dots 0$
 $0 \dots 2 \quad 2 \dots 1$
 $1 \dots 0$
 $1 \dots 2$

Continue l_1 :-

$1 \dots 0$
 $2 \dots 0$
 $2 \dots 1$

do-while vs Continue :- (Very hot combination)

x = 1

Ex:-

```

int x=0;
do
{
    x++;
    S.o.println(x);
    if (++x < 5)
        continue;
}

```

```

    x++;           1
    S.o.println(x);  2
}
while (++x < 10);  3

```

(i)

```

x = 0           0
                1 < 5
x++           1
x < 5         2
x++           3
x < 5         4
x++           5
x < 5         6
x++           7
x < 5         8
x++           9
x < 5         10

```

Imp Note!

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

Ex:-

- ① if (true)

```

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

```

O/P! - Hello

② while(true)

```

    {
        S.o.println("Hello");
    }
    {
        S.o.println("Hi");
    }

```

→ C.E :-

Unreachable Statement

2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
679
679
680
681
682
683
684
685
686
687
688
689
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
719
719
720
721
722
723
724
725
726
727
728
729
729
730
731
732
733
734
735
736
737
738
739
739
740
741
742
743
744
745
746
747
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
769
769
770
771
772
773
774
775
776
777
778
779
779
780
781
782
783
784
785
786
787
788
789
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
809
810
811
812
813
814
815
816
817
818
819
819
820
821
822
823
824
825
826
827
828
829
829
830
831
832
833
834
835
836
837
838
839
839
840
841
842
843
844
845
846
847
848
849
849
850
851
852
853
854
855
856
857
858
859
859
860
861
862
863
864
865
866
867
868
869
869
870
871
872
873
874
875
876
877
878
879
879
880
881
882
883
884
885
886
887
888
889
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
909
910
911
912
913
914
915
916
917
918
919
919
920
921
922
923
924
925
926
927
928
929
929
930
931
932
933
934
935
936
937
938
939
939
940
941
942
943
944
945
946
947
948
949
949
950
951
952
953
954
955
956
957
958
959
959
960
961
962
963
964
965
966
967
968
969
969
970
971
972
973
974
975
976
977
978
979
979
980
981
982
983
984
985
986
987
988
989
989
990
991
992
993
994
995
996
997
998
999
1000



Declarations & Access Modifiers

① Java Source file structure (1 - 9)

Package :-

② Class modifiers (10 - 12)

③ member modifiers (13 - 23)

* ④ Interfaces (24 - 31)

Java Source file Structure :-

- A Java program Can Contain any no. of classes but atmost one class Can be declared as the public. if there is a public class the name of the program & name of public class must be matched otherwise we will get CompiletimeError.
- If there is no public class Then we can use any name as Java Source file name, There are no restrictions.

Ex :- class A

{

}

Class B

{

}

Cass C

{

}

Save: Sri.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
{
 public static void main(String[] args)
 {
 System.out.println("A class main method");
 }
}

Class B
{
 public static void main(String[] args)
 {
 System.out.println("B class main method");
 }
}

Class C

{

P·S·v·m(Staing[] args)

{

S·o·pln("C class main method")

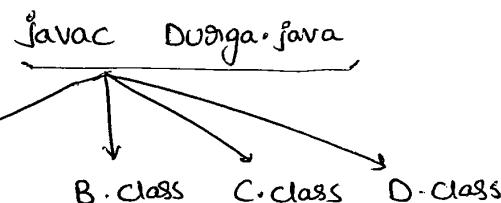
{

Class D

{

}

Save ⇒ Durga.java



① java A ←

A class main method

② java B ←

B class main method

③ java C ←

C class main method

④ java D ←

R.E.: NoSuchMethodError: main

⑤ java Durga ←

R.E.: NOClassDefFoundError: Durga

Note :-

→ 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 we are using import statement it is not required to use fully qualified name hence it reduces imports 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 file will be loaded at the time of import statement, in the next lines of code whenever we are loading 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 5:-

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 6:-

```
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,

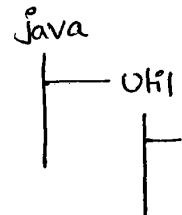
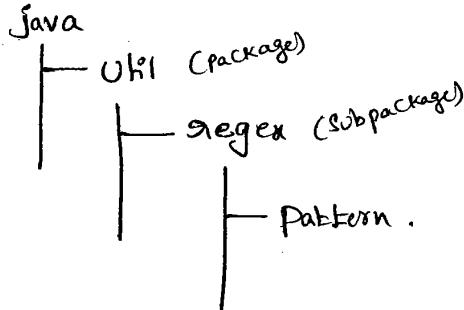
→ Order See above

Date is available in both Util
List is available in both Sql

Case 7 :-

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

Ex:-



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

- * ① import java.*;
- * ② import java.util.*;
- ✓ ③ import java.util.regex.*;
- ✓ ④ import java.util.regex.Pattern;

Case 8 :-

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

- ① java.lang package.
- ② Java default package (current working directory).

Case 9 :-

→ Import statement is totally compiletime issue if no of imports increases then 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.

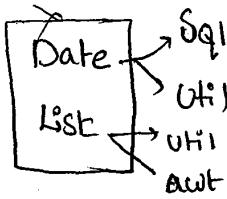
Ex:-

Without Static import

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



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

Class Test

{

p. Static String name = "xyz";

}

Test.name.length();

↙

It is a class
name

↓ It is a method
present in String class

Static variable

Present in Test class

of the type String

Class System

{

Static PrintStream Out;

}

System.out.println();

↙

↓ It is a method
present in PrintStream
class.

It is a static

variable of

type PrintStream

Present in System
class

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. p( 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 &147483647.

→ If we are Commenting Lines ① & ② 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 imports.

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.math;`

✓ ⑤ `import static java.lang.math.*;`

X ⑥ `import static java.lang.math.Sqrt();` → problems

✓ ⑦ `import static java.lang.math.Sqrt;`

Normal import Vs Static import:-

→ we can use normal import to import classes & interfaces

of a package. whenever 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. whenever we are using static import then it is not required to class name to access static members we can access directly.

Packages

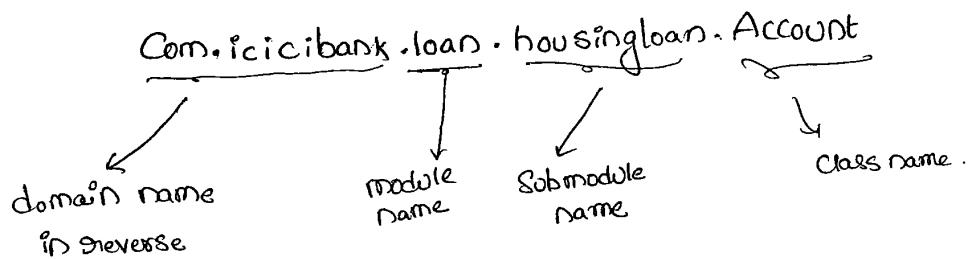
9846 classes are
there in Java
according to 1.6v

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.dreamjobs.itjobs;
```

```
public class HdJobs
```

```
{
```

```
    p.s.v.m(estring) args
```

```
{
```

```
    s.o.println("Getting jobs is very easy");
```

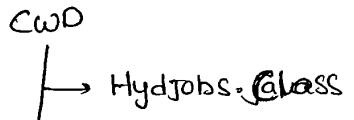
```
}
```

```
}
```

① `javac HydJobs.java`

71

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

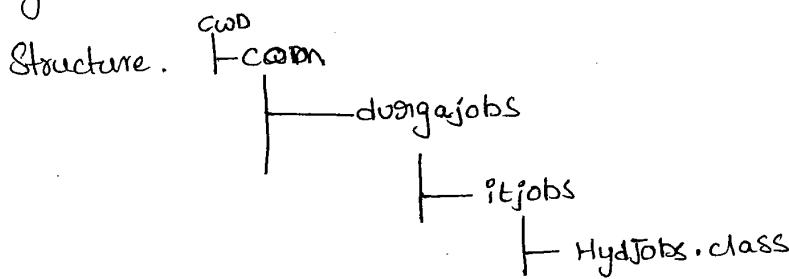


② `javac -d . HydJobs.java`

→ Destination
to place generated
class files

Current
working
directory

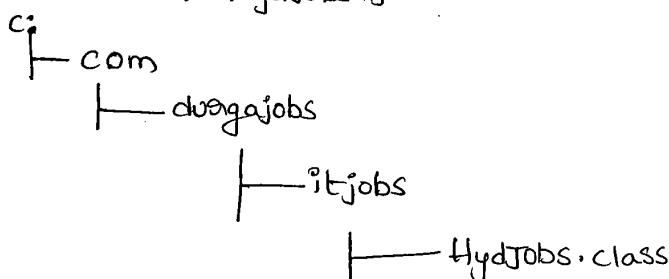
→ generated class file will be placed into Corresponding package



→ 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 Compiletime Error.

Run

→ Java com.durgajobs.itjobs.HydJobs ←

Op:- 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.

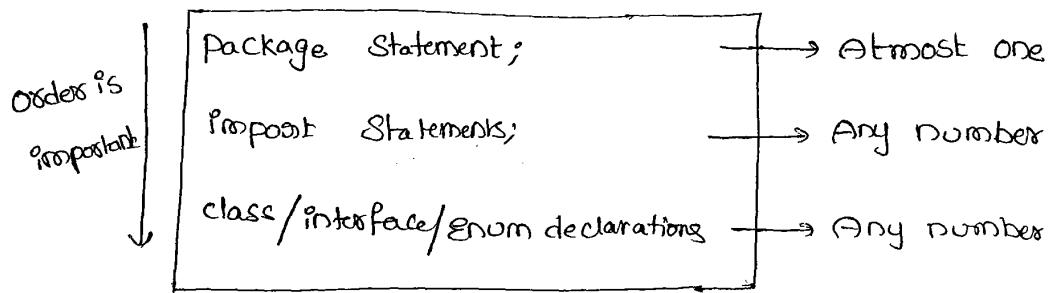
Ex:- ✓ package pack1;
→ package pack2; ←
class A
|
|
C.E:- Class, interface or enum expected.

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

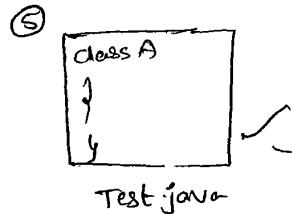
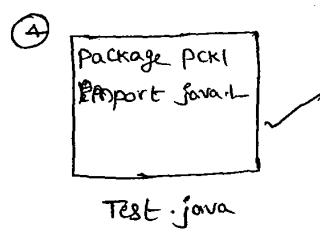
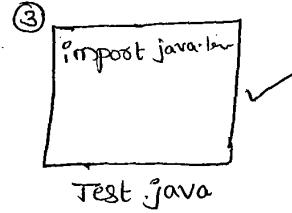
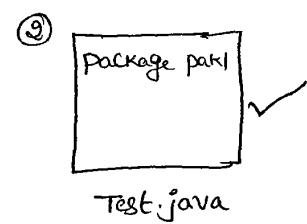
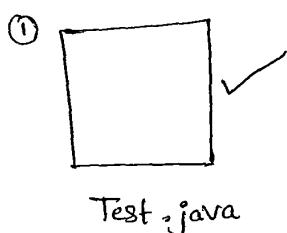
Ex:- ✓ import java.util.*;
→ package pack1;
class A
|
|
C.E:- Class, interface or enum Expected.

→ The proper structure of a Java Source file is

72



→ The following are Valid Java programs.



→ An Empty Source file is a Valid Java program.

** Class modifiers **

→ whenever we are writing 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 CompiletimeError.

Saying "modifier xxxxxxxx not allowed here".

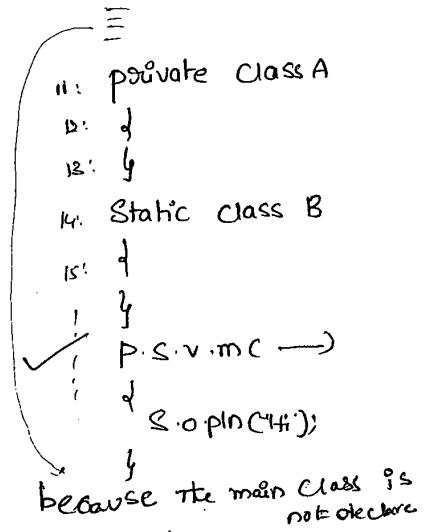
Ex:- Private class Test

```
    |
    p.s.v.m(—)
    |
    int x=0;
    for(int y=0; y<3; y++)
        |
        x=x+y;
        |
        S.O.Println(x);
```

C.E! modifier private not allowed
here

→ But for the Inner classes the following modifiers are allowed 73

- (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.

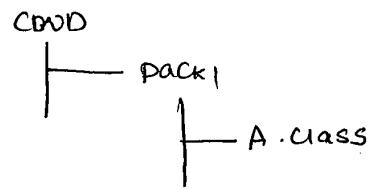
Public classes :-

→ If a Class declared as the public then we can access that class from any where.

Ex:-

```
package pack1;  
public class A  
{  
    public void m1()  
    {  
        System.out.println("Hello");  
    }  
}
```

javac -d . A.java



```

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

```

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 | zumba | attara");
}
|
}

```

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

```

|           |
|           |
|           X

```

Ex:- final class P
|
|
|
Class C extends P
|
|

C.E!:- Can't inherit from final p.

- Every method present inside a final class is always final 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 compulsorily ends with ; .

Ex:- X) public abstract void m1(); }

✓) public abstract void m2();

→ Child classes are responsible to provide implementation for parent class abstract methods.

Exo:-

abstract class Vehicle

|

public abstract int getNoOfWheels();

}

Class Bus extends Vehicle

|

public int getNoOfWheels()

|

return 6;

}

Class Auto extends Vehicle

|

public int getNoOfWheels()

|

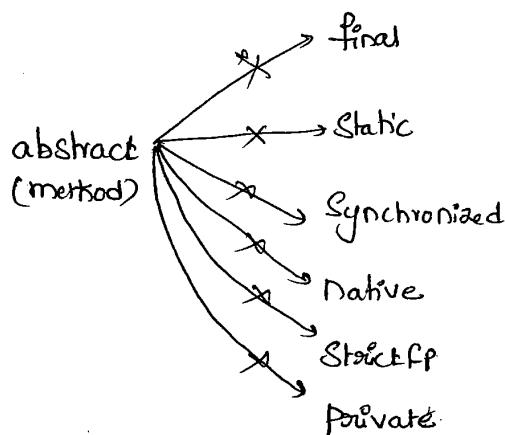
return 3;

}

→ 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

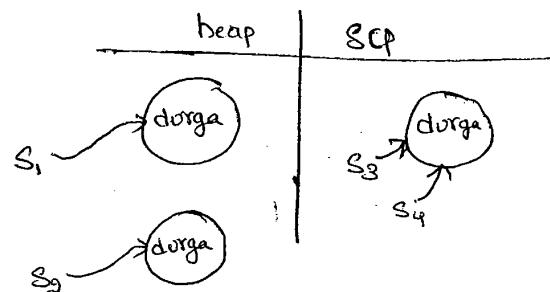
Test t = new Test();

Ex:- String $s_1 = \text{new String}(\text{"durga"});$

String $s_2 = \text{new String}(\text{"durga"});$

String $s_3 = \text{"durga"};$

String $s_4 = \text{"durga"};$



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

- ✓ 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 (r.u)
- 17) I18N
- 18) enum
- 19) development

~~Dell~~

chaithanya@del.com

Sathishdwivedi@ " "

29444524

Chait

chaithanya-Anumanchi@del.com.

ON punjabjobsInfo 9870867070

S

{

abstract class Vs abstract method :-

77

- If a class Contains atleast One abstract method then Compulsory That class should be declared as abstract otherwise we will get ~~CompiletimeError~~. because, The implementation is not Complete & hence We Can't Create an object.
- Even though This class doesnot Contain any abstract method still we can declare the class as abstract. i.e, abstract class Can Contain Zero "0" no.of abstract method.

Ex:- 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 override 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 m2() in Test

→ We can handle these compiletime errors either by declaring SubTest as abstract or by providing implementation for m2().

Note:-

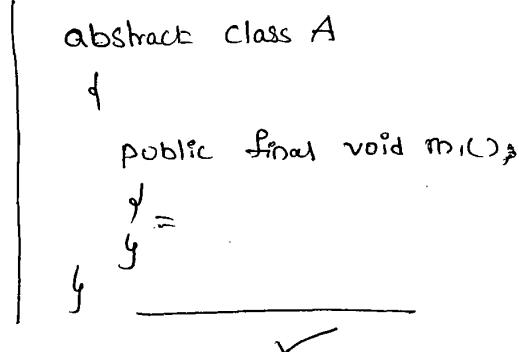
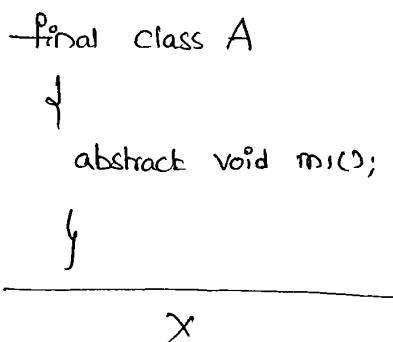
→ The usage of abstract methods, abstract class & interfaces are recommended & it is always good programming practice.

Abstract Vs Final :-

→ abstract methods we have to override in child classes to provide implementation, 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) modifier :- (strictfloatingpoint)

- Strictfp is the modifier applicable for methods & classes but not for variables.
- if a method declared as Strictfp all floating point 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 Structfp void m1(); X (invalid)

Member (variables & methods) modifiers :-

① public members :-

→ If we declare a member as public then we can access that member from anywhere but corresponding class should be visible (public) i.e., before checking member visibility we have to check class visibility.

Ex:-

```
Package pack1;  
class A  
{  
    public void m1()  
    {  
        System.out.println("Hi");  
    }  
}
```

```
Package pack2;  
import pack1.A;  
Class B  
{  
    p.s.v.m(____)  
    {  
        A a = new A();  
        a.m1();  
    }  
}
```

→ Even though m1() method is public, we can't access m1() from outside of pack1 because the corresponding class A is not declared as public. If both are public then only we can access.

② default members :-

→ If a member declared as the default, then we can access that member only within 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 m()
           {
               System.out.println("The most misunderstood modifier in java");
           }
       }
   
```

Class B extends A

```

        |
        P.B.m()
        |
        { }
   
```

✓ A a = new A();
✓ I a.m();

→ The most restricted modifier
is "private"

$\checkmark B \ b = \text{new } B_U$

→ The most accessible modifier
is "public"

$$\checkmark A \quad a_i = \text{new } BC$$

→ private < default < protected
< public

Package pack2;

→ The recommended modifier for
variables is private

Impost packl. A;

→ The Recommended modifier for methods is public

public class C extends A

1

P-S-V-M(—)

۲

$A \cdot a = \text{new } A()$

\times $a \cdot m, (1);$

$c \leftarrow \text{new } C()$

✓ C.M.(S)

$$A - a_1 = \text{new } C()$$

$a_1, m_1(\cdot)$;

1

pack 1
A | ~~A~~ pack 2
└ Protected void m, u

package 3

→ D extends B

→ The most restricted

* private < default < protected < public

80

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;
}
```

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

Rule:-

- ① 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;
    |
    y     ✓
  
```

Class Test

```

    |
    final static int x;
    |
    y   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

✓ {
 final static int x=10;
 }
 }

② Inside Static Block

e.g.- Class Test

✓ {
 final static int x;
 Static
 {
 x=10;
 }
 }
 }

→ 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");
}
}
%
```

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

```

|
P.S.V.M()
|
final int x;
System.out.println("Hello Sou");
}
}
%
```

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

Ex:- Class Test

```

|
P.S.V.M()
|
public int x=10; X
private int x=20; X
static int x=60; X
protected int x=30; X
final int x=40; ✓
}
}
```

- formal parameters of a method Simply access as Local variables of that method. hence,a formal parameter can be declared as final.
- If we declare a formal parameter as final within the method we Can't change its value otherwise we will get Compiletime Error.

Ex:-

Class Test



P.S.V.m()



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.println(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;

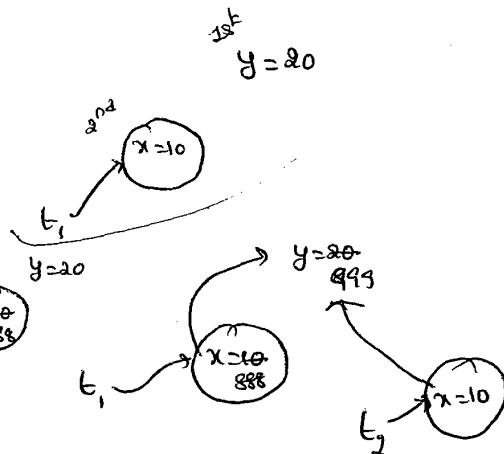
P.S.V.M(—)



Test t₁=new Test();

t₁.x = 888;

t₁.y = 999;



Test t₂=new Test();

S.o.println(t₂.x + " --- " + t₂.y);



10

999

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 CompiletimeError.

Q) Consider the following declarations

I. int x=10;

II. Static int x=10;

III. Public void m1()

↓
S.o.println(x);



IV. Public static void m1()

↓
S.o.println(x);



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

✓ A) I & III

✗ B) I & IV. L.E! - 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 with in the same class we can declare 2 main methods with different arguments

Ex:- Class Test

↓

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

↓

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

↓

public static void main(int[] args)

↓

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

↓

Output: String[]

→ But JVM also

→ But Jvm always Call 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 execute.

Ex:- Class P

```

    |
    P . s . v . m ( String [ ] args )
    |
    S . o . p l n ( " Parent Class " );
    |
    }
```

Class C extends P

```

    |
    }
```

JavaC p . java

P . class

C . 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.

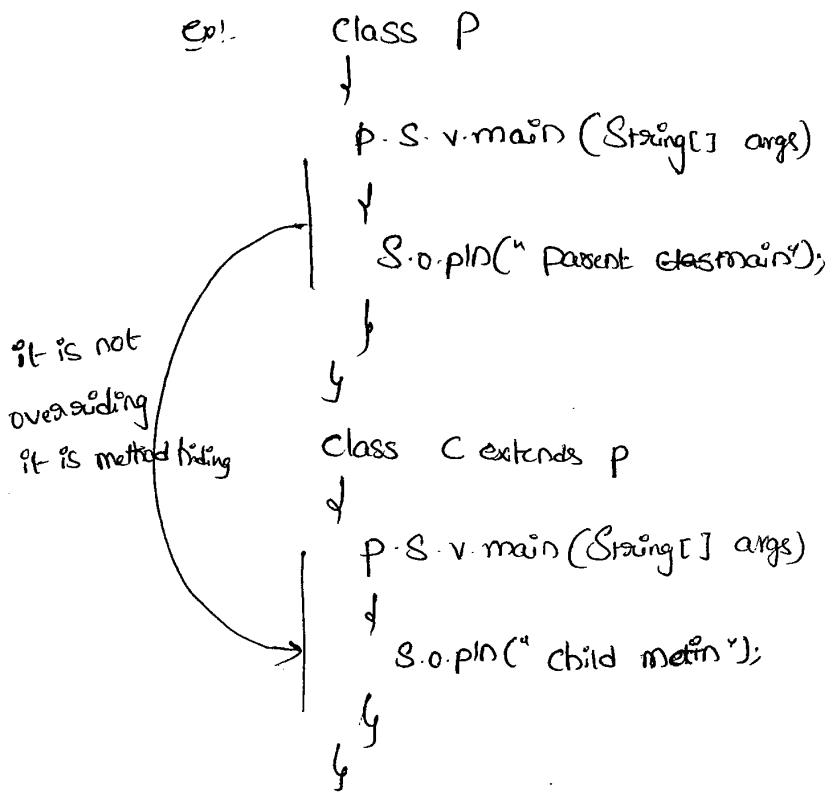
Ex:-

Class P

```

    |
    }
```

P . s . v . m (→)



Java>C P.java

P.class C.class

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.
 - ② To use already existing legacy non-Java code.

Pseudo Code :-

- To use native keyword

Ex:-

```
class Native
```



Static

Scope

③ Load native library → System.loadLibrary("native Library")



④ Declare public native void m();

a native method

```
Class Child
```



p.s.v.m()



⑤ Invoke a Native n = 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
 |
 d

 Public Native void m1()

 |
 g
 X

 } 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 it can resolve data inconsistency problems. But the main dis-advantage of synchronized keyword is it increases waiting time of thread and affects 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.

- 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 got finalized just before terminating the thread the master copy value will be updated with local stable value.
- The main advantage of volatile keyword is we can ~~can~~ resolve data inconsistency problems.
- But the main disadvantage of volatile keyword is, creating & maintaining a separate copy for every thread, increases complexity of the program & 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 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 are native & synchronized.
- The modifiers which are applicable for top level classes, methods & variables are public, <default>, final.

→ The modifiers which are applicable for inner classes but not for
Outer classes are private, protected, static

Interfaces

(1) Introduction

- (2) Interface declaration & Implementation
 - (a) extends vs implements.
- (3) Interface methods
- (4) Interface Variables
- (5) Interface Naming Conflicts
 - (1) method Naming Conflicts
 - (2) Variable " "
- (6) Marker Interface
- (7) Adapter class
- (8) Abstract Class Vs Concrete Class vs Interface.
- (9) diff. b/w abstract class & interface

Interface:-

- 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.
- Within the Interface we can't write any implementation because it has to highlight just the Set of Services what we are offering or what you are expecting. Hence every method present inside interface should be abstract. Due to this interface is Considered as 100% pure Abstract class

What is an Interface:-

- Any Service requirement Specification (SRS) or Any Contract b/w Client & Service provider (or) 100% pure abstract class is nothing but an Interface.
- The main Advantages of Interfaces are.

- (i) we can achieve security, because we are not highlighting our internal implementation.
- (ii) Enhancement will become very easy, because without effecting outside person we can change our internal implementation.
- (iii) Two different Systems can communicate via Interface
(A Java application can talk with Mainframe System through Interface).

Declaration & Implementation of an Interface :-

→ We can declare an Interface by using Interface keyword, we can implement an Interface by using implements keyword.

Ex:-

```

interface Interf
{
    void m1(); // by default public abstract void m1();
    void m2();
}

abstract class ServiceProvider implements Interf
{
    public void m1()
    {
        ...
    }
}

```

→ If a class implements an interface 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.

Extends Vs implements :-

1. A class can extend only one class at a time.
2. A class can implement any no. of interfaces at a time.
3. A class can extend a class and can implement any no. of interfaces simultaneously.
4. An interface can extend any no. of interfaces at a time.

Ex:- interface A

↓
g

interface B

↓
g

interface C extends A, B

↓
g

Q) Which of the following is True?

- (1) A class can extend any no. of classes at a time. X
- (2) A class can implement only one Interface at a time. X
- (3) A class can extend a class and 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

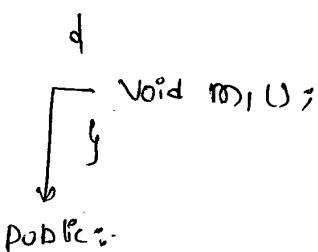
Z → interfaces

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



→ 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 modifiers are not applicable for interface methods.

- | | | |
|---------------|---|------------------|
| (1) private | X | (5) static |
| (2) protected | | (6) static fp |
| (3) <default> | | (7) synchronized |
| (4) final | | (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 int x=10;
- 7) public final int x=10;

8) Static final int $x=10;$

91
27

→ 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 Compulsory one should perform initialization at the time of declaration only otherwise will get compile time error.

④ Interface Interview

) ↓
) int $x;$ X C.E :- = Expected .
)

) → which of the following variable declarations are allowed inside interface.

- ⇒ (1) int $x=10;$ ✓ (5) transient int $x=10;$ X
- (2) int $x;$ X (6) volatile int $x=10;$ X
- (3) private int $x=10;$ X
- (4) public int $x=10;$ ✓ (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(); 88

✓

Interface Naming Conflicts :-

① Method naming Conflicts :-

Case1:-

→ If Two interfaces Contains a method with Same Signature & Same return type in the implementation class we can provide implementation for only one method.

Ex:-

interface Left

↓

public void m1();

↓

interface Right

↓

public void m1();

↓

Class Test implements Left, Right

↓

public void m1()

↓

↓

✓

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.

Ex:-

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

↓

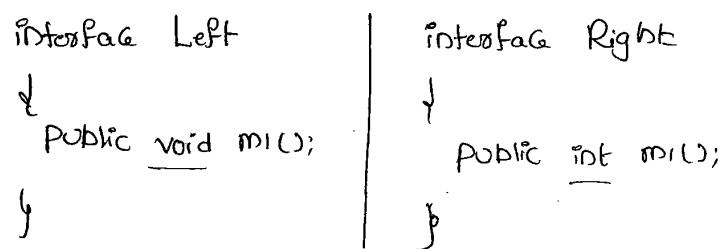
↓

↓

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 :-



→ 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.

A) 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

93

{

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

* Marked 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 marked interface or Tag interface or ability interface.

Ex:- Serializable, Clonable, RandomAccess, SingleThreadMode.

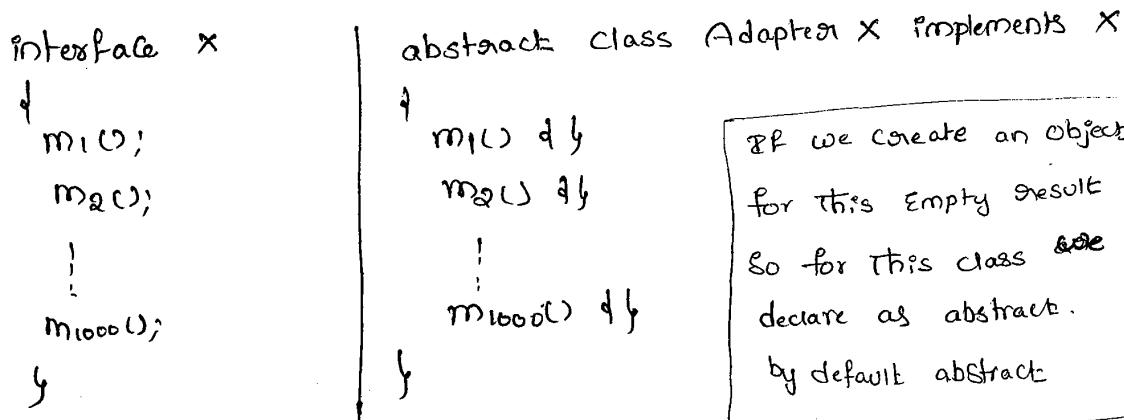
→ These interfaces are marked from some ability.

Ex:- By implementing Serializable interface we can send object across the Net and we can save state of object to a file. This extra ability is provided through ^{extra} Serializable interface.

- Ex:- By implementing Cloneable interface our Object will be in a position to provide exactly duplicate object.
- Q) Marker interface wont contain any method then how the objects will get that special ability?
- A) JVM is responsible to provide required ability in marker interfaces.
- Q) Why JVM is providing required ability in marker interface?
- A) To reduce complexity of the 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, or interface only with empty implementation.



→ If we implement an interface directly ~~we~~ 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 for required method but not all this approach reduce length of the code & improves readability.

⇒ Class Test extends Adapter X

```

    {
        m4() { }
        ==>
        {
            m100() { }
        }
    }
```

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 modifier i.e, we can use any modifier.

→ 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.

→ 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.

96

卷之三

وَمِنْهُمْ مَنْ يَرْجُو
أَنَّ الْأَرْضَ
يُنَزَّلُ عَلَيْهِ
مِنْ كُلِّ ثَمَنٍ
وَمِنْهُمْ مَنْ يَرْجُو
أَنَّ اللَّهَ
يُنَزِّلَ
عَلَيْهِ
مِنْ فَضْلِهِ
مَا لَمْ يَرَ

28/4/11

OOPS Concept

xox

- 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 persons 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 effecting outside person we can change our internal implementation hence Enhancement will become very easy.

→ The main disadvantage of Encapsulation is it increases the length of the code & slows down execution.

4) Tightly Encapsulated class:-

→ A class is said to be tightly encapsulated iff every data member declared as the private.

→ 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;
}
  
```

Ex:- 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;
    |
    | 
  
```

③ It improves modularity of the application. meaning?

3) Encapsulation :-

→ Encapsulating data & corresponding methods (behaviour) into a single module is called "Encapsulation".

→ If any Java class 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)

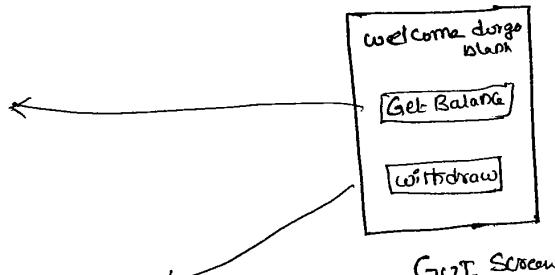
↓

// validate user

this.balance = balance;

↓

↓



GUI Screen

→ Hiding data behind methods is the Central Concept of Encapsulation

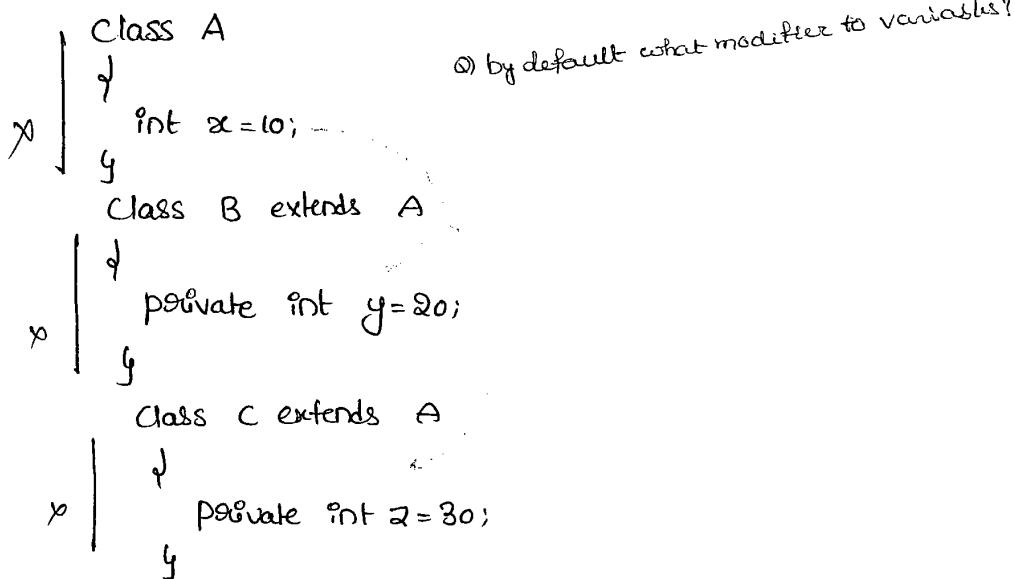
→ The main advantages of Encapsulation are ① We can achieve Security.

② Enchancement 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 p₁ = new C();

p₁.m₁(); ✓

p₁.m₂(); X → C.E.

* Case 4: C c₁ = new P(); X C.E. incompatible types
found : P
Required : C

Conclusion:-

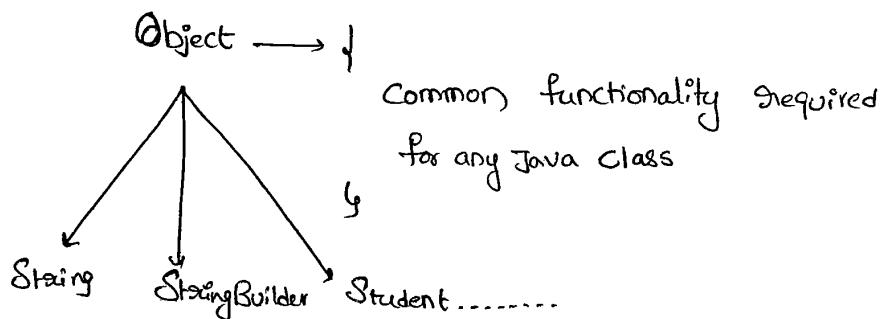
① whatever the parent class has by default available to the child. Hence ^{with them} child class reference we can call both parent & child class methods.

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 cant call child specific methods.

- ③ Parent class reference can be used to hold child class objects by using that reference we can call only parent class methods but we can't call child specific methods.
- ④ We can't use child class reference to hold parent class objects.

Ex:-

- ① The common functionality which is required for any Java classes is defined in Object class and by keeping that class as Super class its functionality by default available to every Java classes.

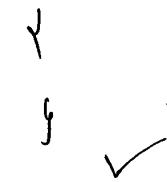


- Ex:- the common functionality which is required for all Exceptions & Errors is defined in Throwable class as Throwable is parent for all Exceptions & Errors, its functionality will be available automatically to every child not required to override.
- Q) Do 'Throwable' has 'Object' as parent class?
Ans: Yes
- Java won't provide support for multiple inheritance but through interfaces it is possible.

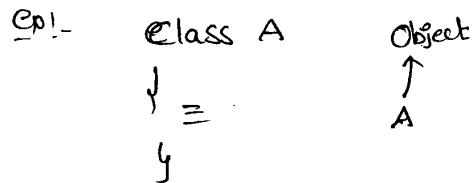
Ex:-

Class A extends B, C

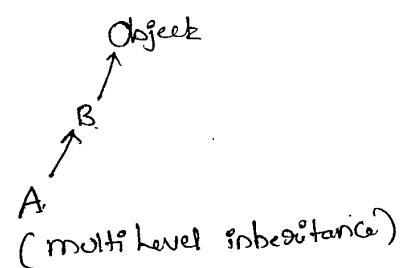
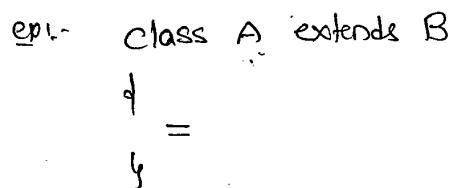
But Interface A extends B, C



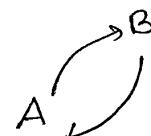
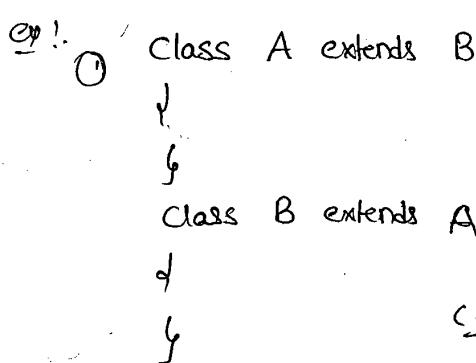
- Every class in Java is the child class of Object.
- If our class doesn't extend any other class then only it is the direct child class of Object.



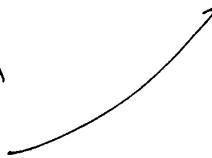
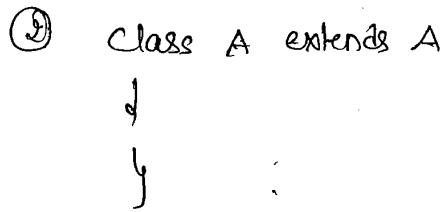
- If our class extend any other class then our class is not directly child class of Object.



- Cyclic inheritance is not allowed in Java



C.E:- Cyclic inheritance involving A



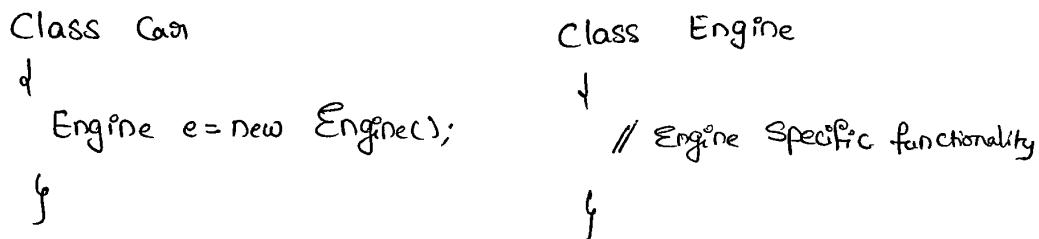
6) Has-A Relationship:-

→ Has-A Relationship is also Known as "Composition or Aggregation".

→ There is no Specific Keyword to Implement Has-A Relationship the Mostly we are Using "new keyword".

→ The main advantage of Has-A Relationship is Reusability or Code Reusability.

Ex:-

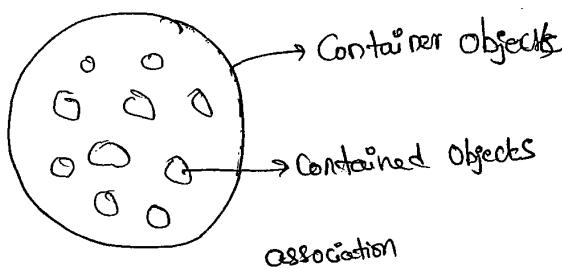


Class Car has Engine reference.

→ 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 Object is destroyed All Contained Objects will be destroyed automatically. i.e., without Existing Container Object there is no chance of existing Contained Object i.e. Container & Contained objects having Strong association



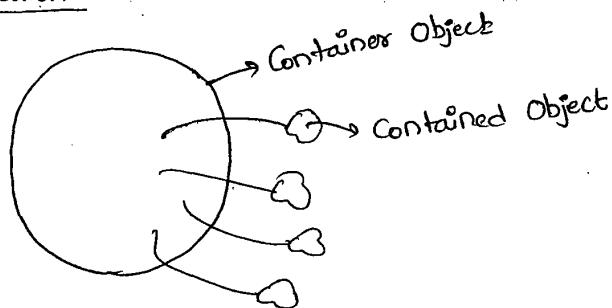
Ex :-

→ University is Composed of Several departments.

→ whenever you are closing University automatically all departments will be closed. The relationship b/w University Object & department object is Strong association which is nothing but Composition.

→ Aggregation :

→ whenever Container Object destroyed, There is no guarantee of destruction of Contained objects i.e., without existing Container object there may be a chance of Existing Contained Object i.e., Container object just maintains References to Contained objects. This relationship is Called WeakAssociation 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.

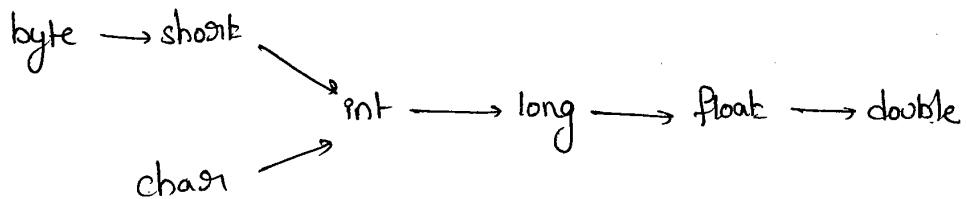
Case1 :-

* Automatic promotion in Overloading :-

- In overloading method resolution, if the matched method with Specified argument type is not available then Compiler won't raise

any error immediately. If it promotes that argument 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 argument 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 ~~feature~~ 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");
}

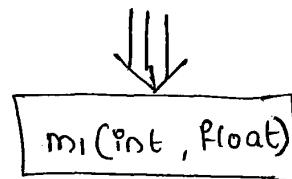
public void m(String[] args)
{
    Test t = new Test();
}
```

Method Signature :-

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

List:-

Ex:- public void m1 (int i, float f)



- 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 Compilation Error.

Ex:- class Test

```

    public void m1(int i)
    {
        public int m1(int i)
        {
            return 10;
        }
    }
  
```

m1(int)
is the method Signature.

Test t = new Test()

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:- $\text{abs}() \rightarrow \text{int}$
 $\text{labs}() \rightarrow \text{long}$
 $\text{fabs}() \rightarrow \text{float}$
 \equiv

→ But in Java two methods having the same name with different arguments is allowed & these methods are considered as overloaded methods.

Ex:- $\text{abs}(\text{int})$
 $\text{abs}(\text{long})$
 $\text{abs}(\text{float})$
 \equiv

→ 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?

Case 2:-

Ex:-

Class Test



public void m1(StoringBuffer sb)



System.out.println("StoringBuffer-args");



public void m1(String s)



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



public String m1()

~~Play~~

By default 'String'
constant of String class
object type
are integral constant of int
floating literal "double"

Test t = new Test();

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

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

X t.m1(null);

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

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

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

`t.m(10.5); X C.E.`

{

{

Cannot find Symbol

Symbol: method m1(double)

location: class Test

Case 2:-

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

Ex:-

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 (—)

{

Object

↑

String

Test t = new Test();

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

`t.m1("durga"); // String-version (Suppose ② statement takes //`

`t.m1(null);? // String-version (String the op is Object)`

{ { } }

// String-version

→ 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 ^{ಒಂಹಾತ ಬದಲಾವಣೆ} 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

↓

So:-

Parent method
return type

↓

Child method
return types

↓

String

Object

↓

Integer

Number

↓

Object

String

↓

Object

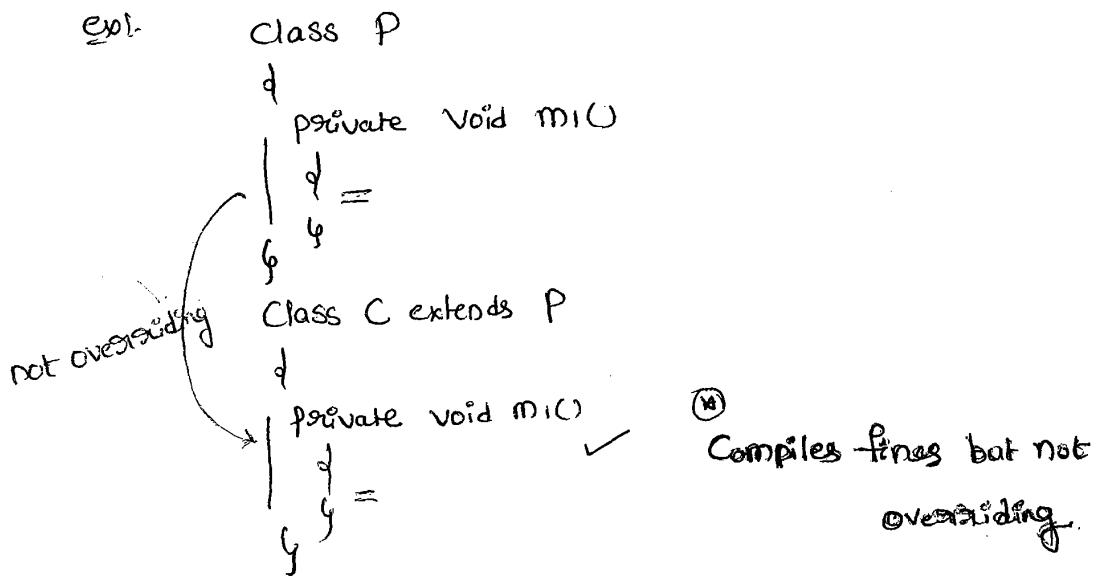
double

↓

int

→ 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 patient class abstract methods we should override in child class to provide implementation.

- ④ → We can override parent class non-abstract method as abstract in child class to stop parent class method implementation availability to the child classes.

10b₁₂

Ex:- Class P

public void p()

4

abstract class C extends P

Public abstract void p();

६

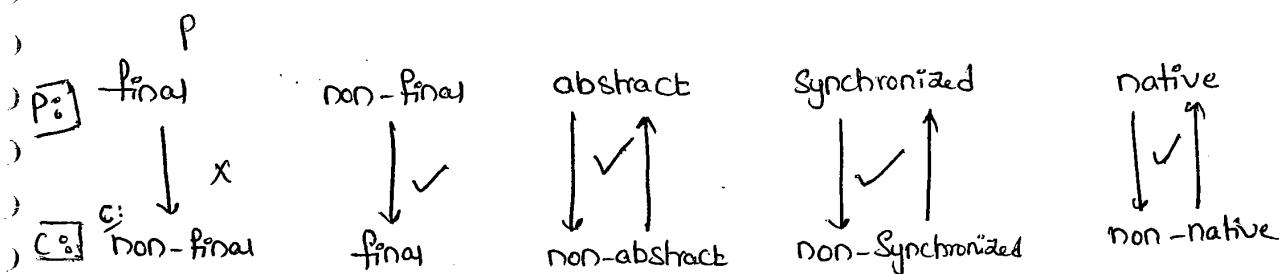
→ The following modifiers won't play any role in structuring in

Overzicht

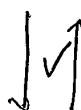
① Native

④ Synchronized

③ stuck pp



Stack Pp

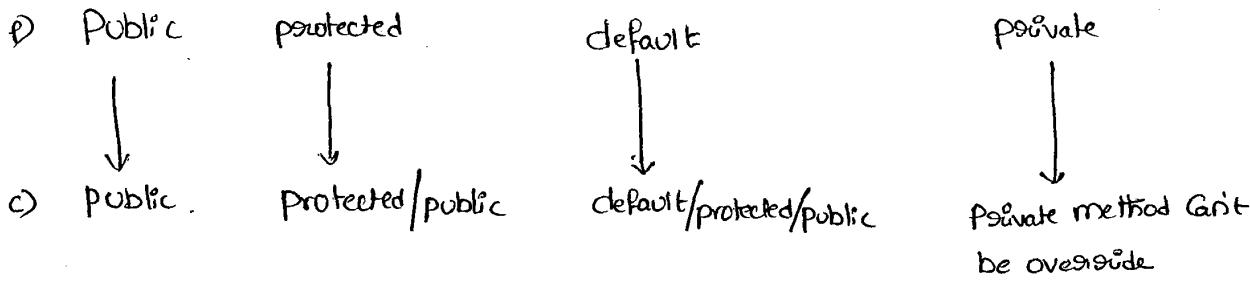


Non - Stückfp

→ 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.

Eg:- Interface Interf

↓

Void m1();

{

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()

{

y

↳

Class C extends P

{

 Public void m1() throws Exception X

y

↳

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()  
    }
```



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

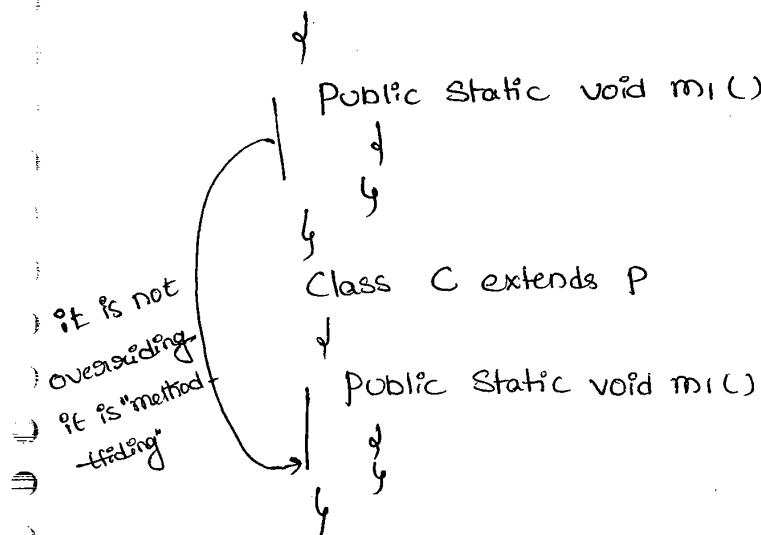
overridden method is static.

→ 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



Method Hiding :-

- All rules of Method Hiding are Exactly Same as Overriding
- Except the following difference.

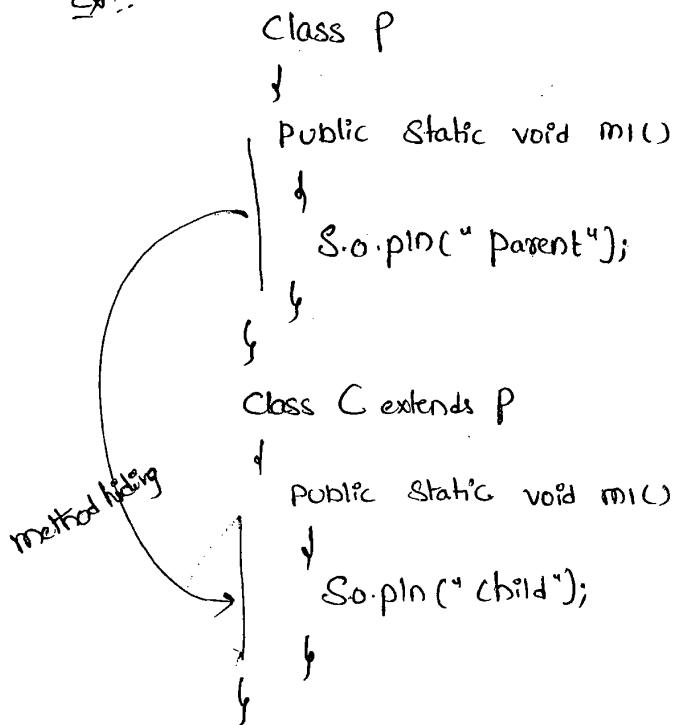
Method Hiding

- Both methods should be static
- Method Resolution takes care by Compiler based on Reference type.
- It is Considered as Compiletime Polymorphism or Static Polymorphism or Early Binding.

Overriding

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

Ex:-



Class Test

p.s.v.m(→)

P p = new P();

p.m1(); → parent

C c = new C();

c.m1(); → child

P p₁ = new C();

p₁.m1(); Parent

}

→ If both methods are non-static then it will become overriding in this

Case the o/p is: Parent

Child

Child

Overriding w.r.t Var-arg methods :-

- We can't override a Var-arg method with general method. If we are trying to override it will become overloading but not overriding.
- A Var-arg method should be overridden with Var-arg method only.

Ex:-

Class P



public void m1(int... i)



System.out.println(" patient");



Class C extends P



public void m1(int i)



System.out.println(" child");



Class Test



p.s.v.m →



P p = new P();

P.m1(10); // Patient

C c = new C();

C.m1(10); // child

P p = new C();

P.m1(10); // parent

overloading
but not
overriding

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

Overriding w.r.t Variables :-

- Overriding Concept is not applicable for variables.
- Variable Resolution always takes Care by Compiler based on Reference type. Runtime object won't to play any role in variable resolution.

Ex:-

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-variant 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 (COP) Static polymorphism (SP) Early binding	Runtime polymorphism (ROP) Dynamic polymorphism (DOP) 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

Overriding

① public void m1 (int i)

overloading

② public void m1 () throws Exception

overloading

③ public static int m1 (double d) throws IOException

C.E 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.

- ↳ 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

e.g. Overloading

Method Hiding

Run-time polymorphism

e.g. Overriding

3 Pillars of OOPS :-

Inheritance
(Reusability)

OOPS

Encapsulation
(Security)

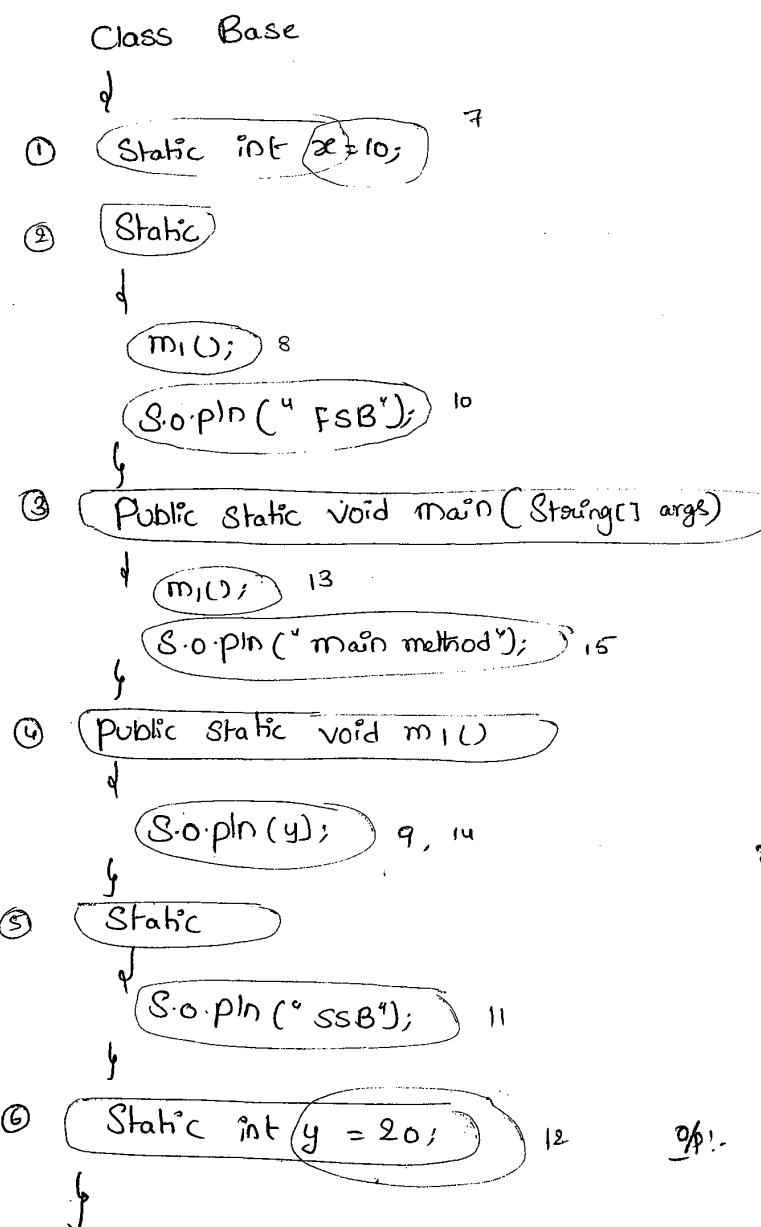
Polymorphism
(Flexibility)

Funny differentiation of polymorphism :-

→ A boy uses the word FRIENDSHIP to starts LOVE, but girl uses the same word to ~~ends~~ Close. Same word but different attitudes. This behaviour is nothing but polymorphism.

Static Control flow :-

Ex:-



x = 0 [R I W O]

y = 0 [R I W O]

x = 10 [R & W]

y = 20 [R & W]

Ans:-
 0
 FSB
 SSB
 20
 main method

Process:-

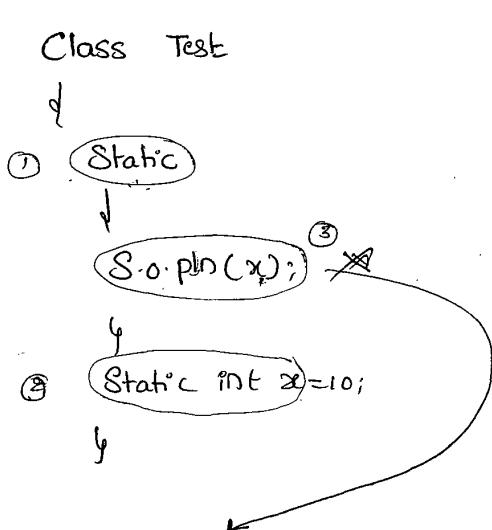
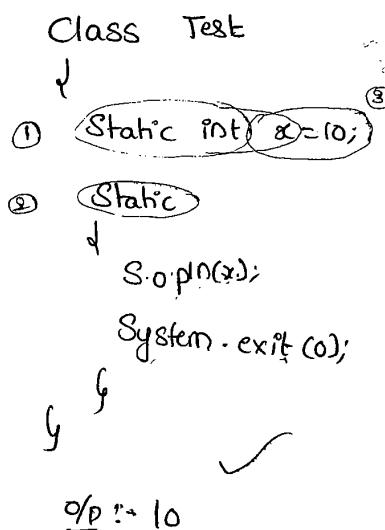
→ whenever we are trying to execute a Java class first that class file should be loaded, at the time class loading the following actions will be performed automatically.

- ① Identification of static members from Top to bottom. (1 to 6)
- ② Execution of static variable assignments & static blocks from top to bottom (7 to 12)
- ③ Execution of main method. (13 to 15)

Read Indirectly write only state (RIWOS)

→ If a variable is in Read indirectly write only state then we can't perform read operation directly otherwise we will get compile-time error saying "Illegal Forward Reference".

Ex:-



C.E:- Illegal Forward Reference.

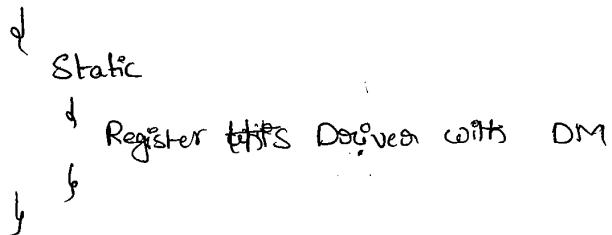
Static block :-

- 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 DriverManager 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.

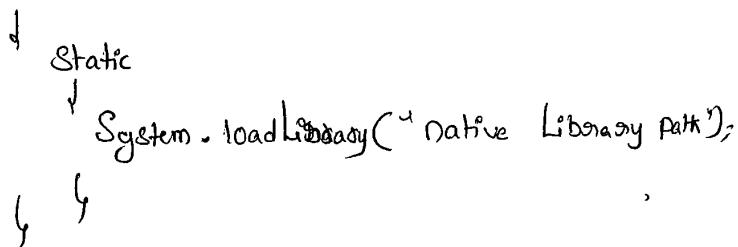
Ex. Class Driver



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



Q) Without using main() method is it possible to print some statements to the Console?

A:- Yes, by using static block

Ex:-

```

class Google
{
    static
    {
        System.out.println("Hello... Boss I can print");
        System.exit(0);
    }
}

```

Q) Without using main() method & static block is it possible to print some statements to the Console?

A). Yes,

Ex:-

```

class Google
{
    static int x = m1();
    public static int m1()
    {
        System.out.println("Hello... I can print");
        System.exit(0);
        return 0;
    }
}

```

O/P:- Hello... I can print.

Ex 2:-

```
class Google
{
    static Google g = new Google();
    Google()
    {
        System.out.println("Hello... I can print.");
        System.exit(0);
    }
}
```

O/P:- Hello... I can print

③

```
class Google
{
    static Google g = new Google();
    {
        System.out.println("Hello... I can print.");
        System.exit(0);
    }
}
```

instance
block

O/P:-

Static Control flow in parents child classes :-

Class Base

↓

① Static int $x=10$; ⑫

② Static

↓

m1(); ⑬

System.out.println(" Base SB"); ⑭

⑤ Public static void main()

↓
m1();

System.out.println(" Basic main");

↓

⑥ Public static void m1()

↓

System.out.println(y); ⑮

↓

⑦ Static int $y=20$; ⑯

↓

Class Derived extends Base

↓

⑧ Static int $i=100$; ⑰

⑨ Static

↓

m2(); ⑱

System.out.println(" DFSB"); ⑲

↓

⑩ Public static void main()

↓
m2(); ⑳

System.out.println(" Derived main"); ㉑

⑨ Public static void m2()

↓

S.o.println(j); ⑯ ⑰

↓

⑩ Static

↓

S.o.p("DSSB"); ⑮

↓

⑪ Static int j = 200; ⑯

↓

> java Derived

O/P:- 0

Base SB

0

DSSB
OSSB

200

Derived main

x=0 [R I W0]

y=0 [R I W0]

i=0 [R I W0]

j=0 [R I W0]

x=10 [R & W]

y=20 [R & W]

i=100 [R & W]

j=200 [R & W]

> java Base

0

Base SB

20

Base main.

PROCESS:-

> javac Derived.java



> java Derived

① Identification of static 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

{

② int ~~x=10~~; ⑨

④ m(); ⑩

⑤ System.out.println("FIB"); ⑫

}

⑥ Parent()

{

⑦ System.out.println("Constructor"); ⑮

}

⑧ public static void main(String[] args)

{

⑨ Parent p = new Parent();

⑩ System.out.println("main");

}

⑪ public void m()

{

⑫ System.out.println(y); ⑯

{

⑬ System.out.println("SIB"); ⑭

{

⑮ int y=20; ⑯

{

x=0 [R I W O]

y=0 [R I W O]

x=10 [R W]

y=20 [R W]

O/P:-

O

FIB

SIB

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; ⑮

⑯ ↓

m1(); ⑯

S.o.pln("parent"); ⑯

{}
↓

④ parent ()

↑

S.o.pln("parent constructor"); ⑯

{}
↓

① public static void main(→)

↓

② Parent p = new Parent();

S.o.println("child main"); ③
↓
Parent

O

parent

Parent Constructor

④ public void m();

↓

S.o.println(y); ⑤
↓

⑥ int y=20; ⑦
↓

Class Child extends Parent

↓

⑧ int i=100; ⑨
↓

⑩ m2(); ⑪
↓

S.o.println("CITB"); ⑫
↓

⑬ Child()
↓

S.o.println("Child Constructor"); ⑯
↓

⑭ Public static void main(→)

↓

⑮ Child c = new Child();

S.o.println("Child main"); ⑰
↓

⑲ public void m2()

↓

S.o.println(j); ⑳
↓

O

CITB

CSITB

Child Constructor

Child main.

```

①   }
S.o.println("CSIIB"); ②
|
int j = 200; ③
}

```

Process :-

- When ever Sequence of we are creating child class object
- The following Sequence of execute events will be performed automatically.
- (i) Identification of instance members from parent to child.
- (ii) Execution of instance variable assignments & instance blocks only in parent class.
- (iii) Execution of parent class Constructor.
- (iv) Execution of instance variable assignments & instance blocks only in child class.
- (v) 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 piece of the code will be executed automatically to perform initialization This piece of code is nothing but constructor. Hence the main objective of Constructor is to perform initialization for the newly created object.

Ex:-

Class Student

{

① int rollno;

② String name;

Student (String name, int rollno)

{

this.name = name;

this.rollno = rollno;

}

Public static void main (String [] args)

{

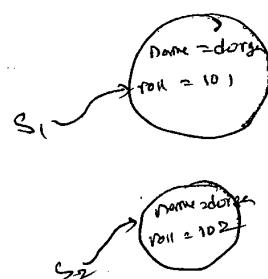
Student s₁ = new Student ("durga", 101);

{

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

}

}



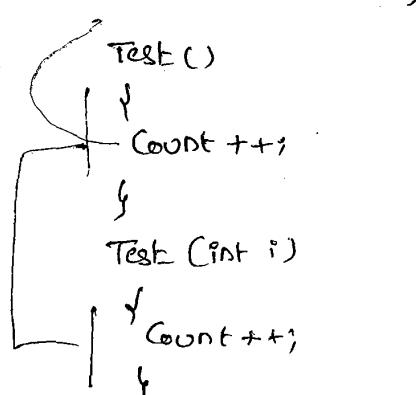
Instance block vs Constructor :-

- At the time of object creation if we want to perform initialization of instance variable then we should go for constructor.
- Other than initialization activity if we want to perform any activity at the time of object creation then we should go for instance block.
- We can't replace constructors with instance block because constructor can take arguments 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

Only once required

If we create instance



p.s.v.m(—)

Test t₁ = new Test();

Test t₂ = new Test(0);

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 Compiletime or Runtime Errors, because Compiler treats it as method.

Ex:- Class Test



void Test()

It is a normal method but not Constructor



It is legal (or 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 / Stack-tp ...

Ex:- Class Test



final Test()



C.E! modifier final is not allowed here.

Singleton classes :-

→ for any java class if we are allowed to Create only one object

Such type of class is called "Singleton Class".

Ex:- Runtime, ActionServlet (Struts 1.x)

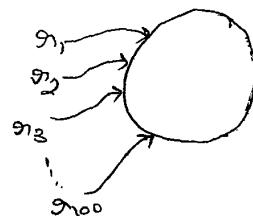
BusinessDelegate (EJB), ServiceLocator (EJB) ---- etc.

→ 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 r₁ = Runtime.getRuntime()

Runtime r₂ = Runtime.getRuntime()
 ! ↳ Class ↳ Static method

Runtime r₁₀₀ = Runtime.getRuntime()



Creation of our own Singleton Class :-

→ We can create our own Singleton classes also for this

We have to use private Constructor & factory method.

Ex:- class Test

 {
 private static Test t;

 private Test()
 }

 public static Test getInstance()
 }

```

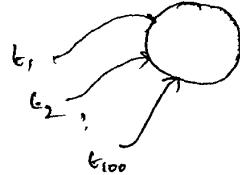
if (t == null)
{
    t = new Test();
}
return t;

```

```

public Object clone()
{
    return this;
}

```



Test t₁ = Test.getInstance();

Test t₂ = Test.getInstance();

⋮

Test t₁₀₀ = Test.getInstance();

Test t₁₀₁ = Test.clone();

factory method:-

→ By using class name if we call any method & return same class object. Then that method is consider as factory method.

Ex:-

→ factory method

Runtime r = Runtime.getRuntime();

Dateformat df = DateFormat.getInstance();

→ factory method.

Test t = Test.getInstance();

→ factory method

→ Similarly we can Create Doubleton, Threbleton ¹²⁰~~Threbleton~~ ²⁶

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, i.e. is a no argument call to Super class Constructor.

```
Test()  
|  
Super();  
|
```

Programmers Code

(1) class Test

↓

{

(2) public class Test

↓

{

}

}

)

(3) Class Test

↓

void Test()

↓

{

(4) Class Test

↓

Test()

↓

{

(5) Class Test

↓

Test()

↓

this(10);

Test(int i)

↓

{

}

Compiler Generated Code

(1) Class Test

↓

Test()

↓

Super();

↓

(2) public class Test

↓

public Test()

↓

Super();

↓

(3) Class Test

↓

Test()

↓

Super();

↓

void Test()

↓

{

}

(4) Class Test

↓

Test()

↓

Super();

↓

(5) Class Test

↓

Test()

↓

this(10);

↓

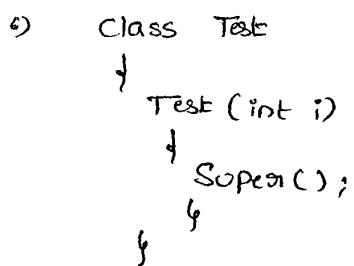
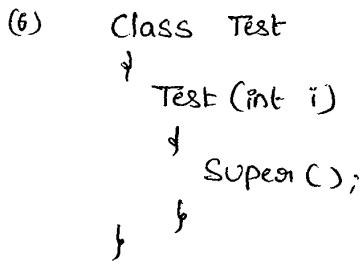
Test(int i)

↓

Super();

↓

121
97

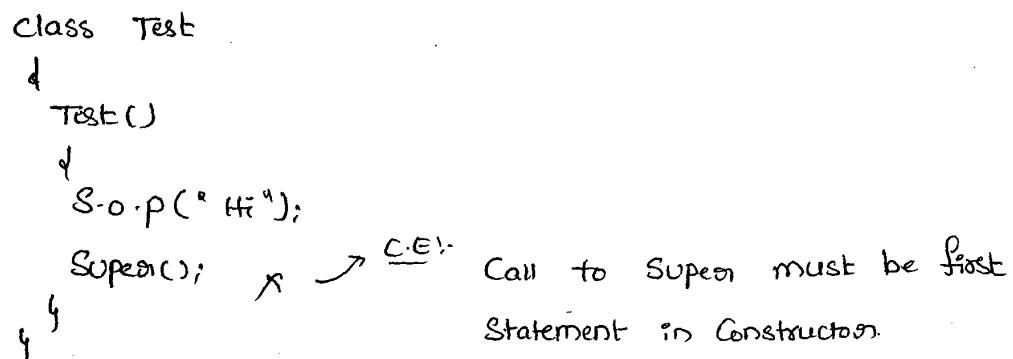


Super & This :-

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

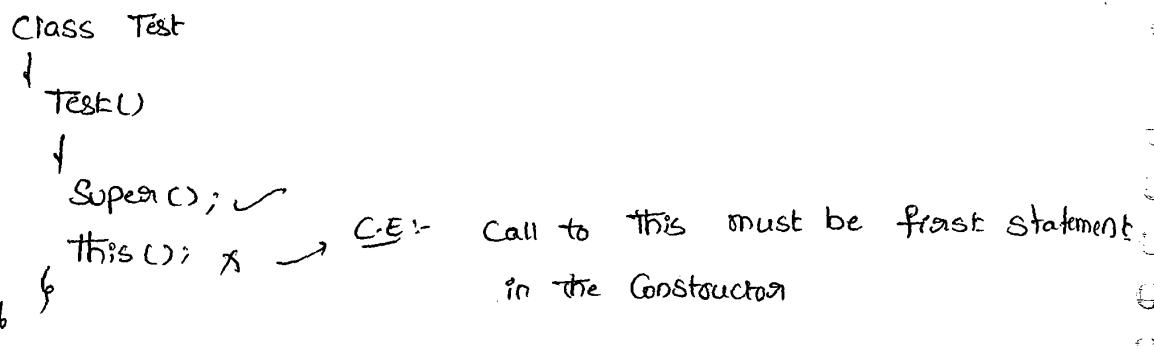
Case(i) :-

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



Case(ii) :-

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



Case(iii) :-

122
28

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

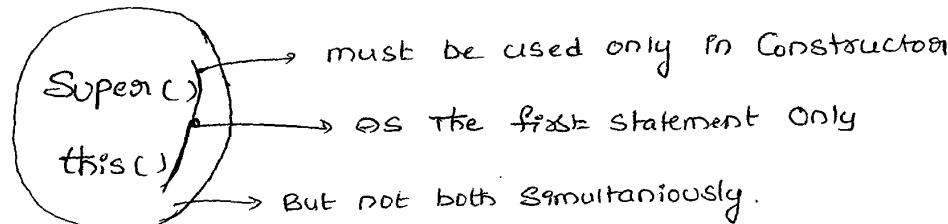
Ex:- Class Test

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

Super(); X → C.E:-

S.o.pn("Hi");

Call to Super must be first statement in the Constructor



this() :- To Call Current class Constructors

Super() :- To Call Parent class Constructors

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

Super()	Super this
(1) These are Constructor Calls	1) These are key words to refer Super & Current class instance members
(2) we should use only in Constructors	2) We can use any where except in static area.

Ex:-

Class Test

↓

P·S·V·M(L)

↓

S·o·pIn(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·pIn("double-args");

↓

Test(int i)

↓

this();

S·o·pIn("int-args");

↓

Test()

↓

S·o·pIn("No-args");

↓

P·S·V·M(—)

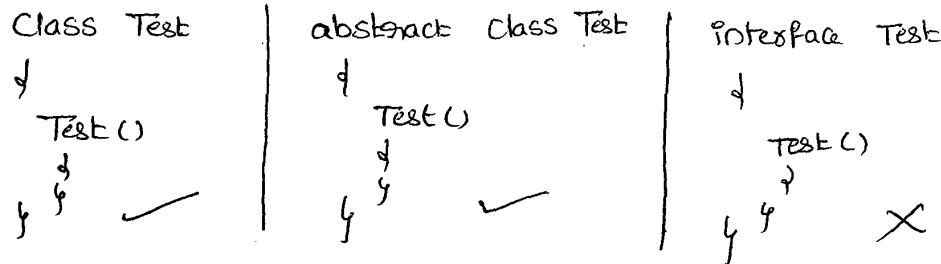
↓

123
29

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.



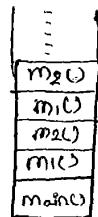
→ Case(i):-

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

e.g:-

```

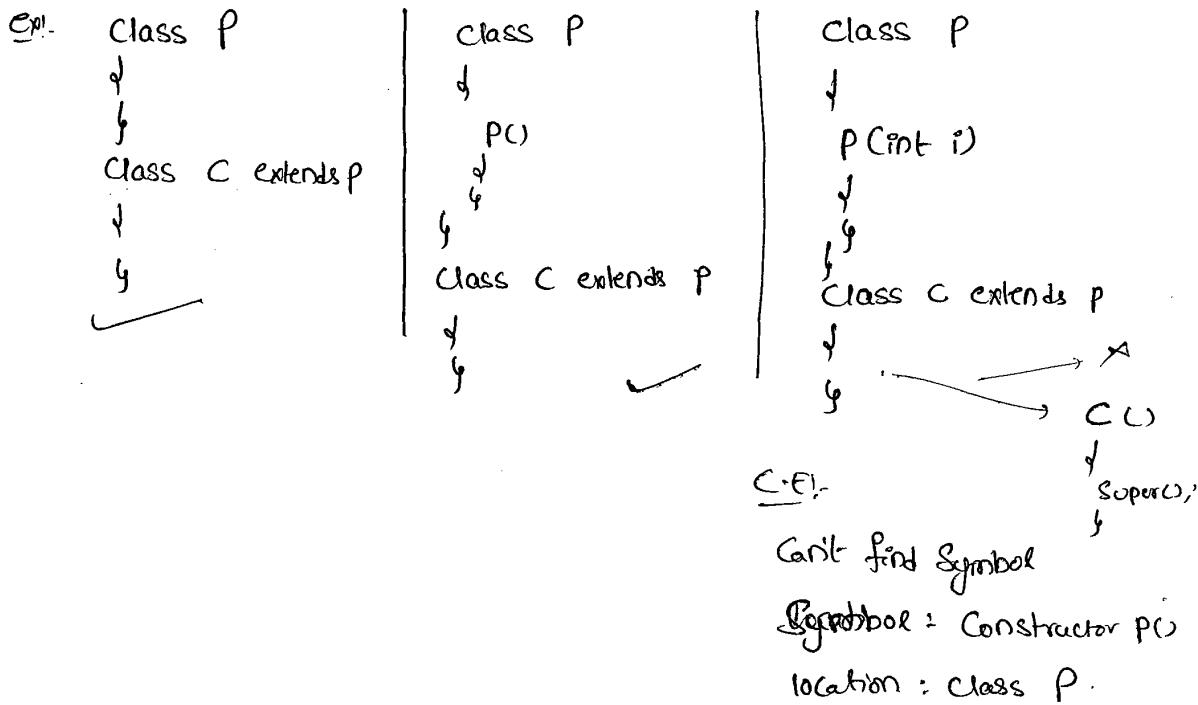
class Test {
  p.s.v.m1()
  | m2();
  |
  p.s.v.m2()
  | m1();
  |
  p.s.v.m();
  | s.o.p("Hello");
  |
  m1(); RE: Stack Overflow Error
  
```



```

class Test
  Test()
  | this(10);
  |
  Test(int i)
  | this();
  |
  p.s.v.m() →
  | s.o.println("Hello");
  |
  C.E! - Recursive Constructor
  invocation.
  
```

Case(ii) :-

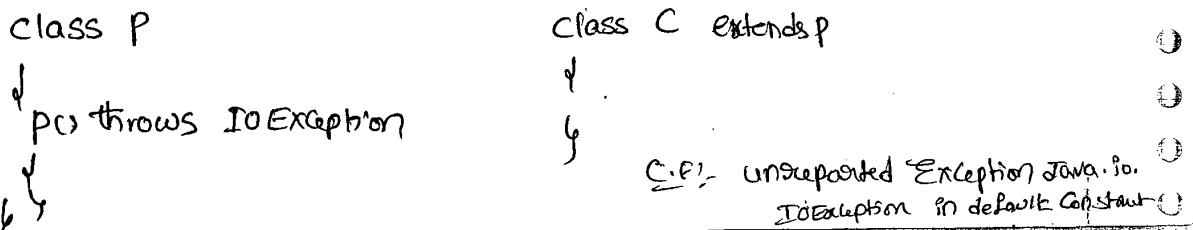


Note:-

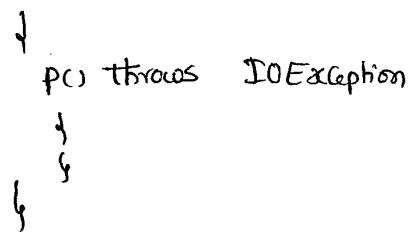
- if the parent class contains some Constructors then while writing child class we have to take special care about Constructors.
- whenever we are writing any argument Constructor it is highly recommended to write no argument Constructor also.

Case(iii) :-

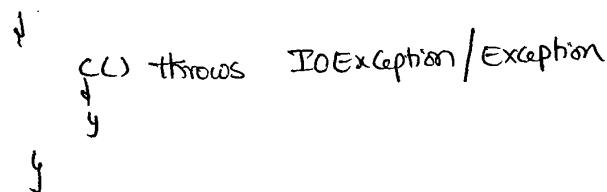
- If parent class constructor throws some checked Exception Compulsory Child class constructor should throw same checked Exception or its parent otherwise the code won't compile.



Ex - Class P



Class C extends P



Q) Which of the following is True?

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

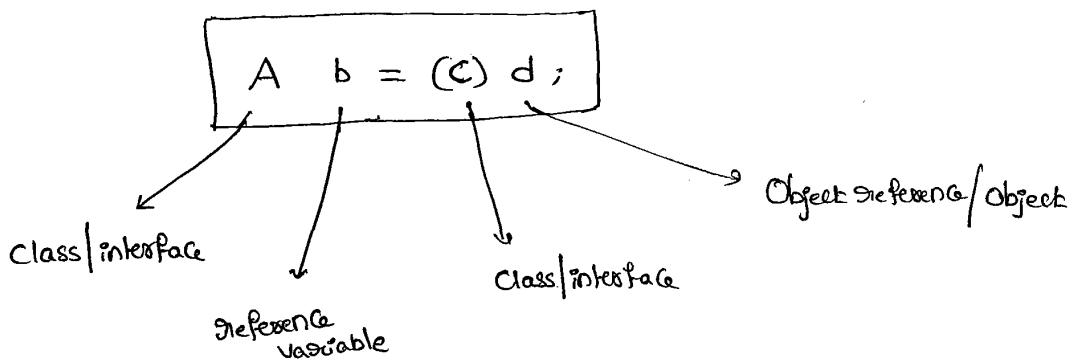
- (1) Interface Can Contains Constructor ✗
- (2) Both overloading & overriding Concepts are applicable for Constructor ✗.
- (3) Inheritance Concept is applicable for Constructor ✗

Type-Casting

Type-Casting:-

- Parent Class Reference Can be used to hold child class object
- Ex:- Parent p = new Child();
- Similarly, interface reference can be used to hold implemented class object.
- Ex:- Runnable r = new Thread();

Syntax:-



Compiler rule(s):-

- 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:-

inconvertible 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:-

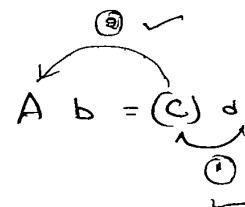
① Object o = new String ("durga");

SB sb = (SB) o; X

Rule ① ✓

② ✓

③ X (R-E):- CCE

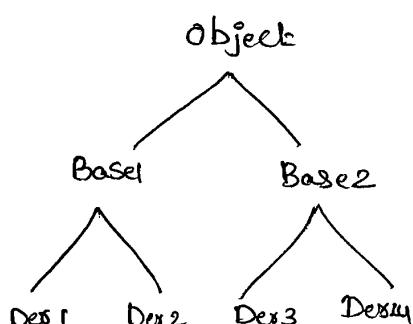


④ Object o = new String ("durga");

String s = (String)o; ✓

Rule ① ✓
② ✓
③ ✗

Ex:-



C.EI:- Inconvertible types

found: Base2

required: Base1

Ex:- ① Base2 b = new Derv();

✓ ② Object o = (Base2) b;

X ③ Object o = (Base1) b;

④ Base2 b1 = (Base2) o;

X ⑤ Base1 b3 = (Der1)(new Derv());

(C.EI:-)

Inconvertible types

found: Der2

required: Der1

18b_{ui}

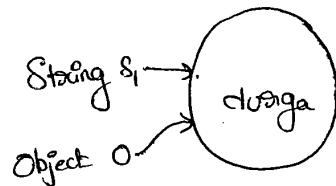
→ Strictly Speaking in Type-Casting just we are Converting only type of object but not underlying object itself

Ex:-

String s₁ = new String("durga");

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();

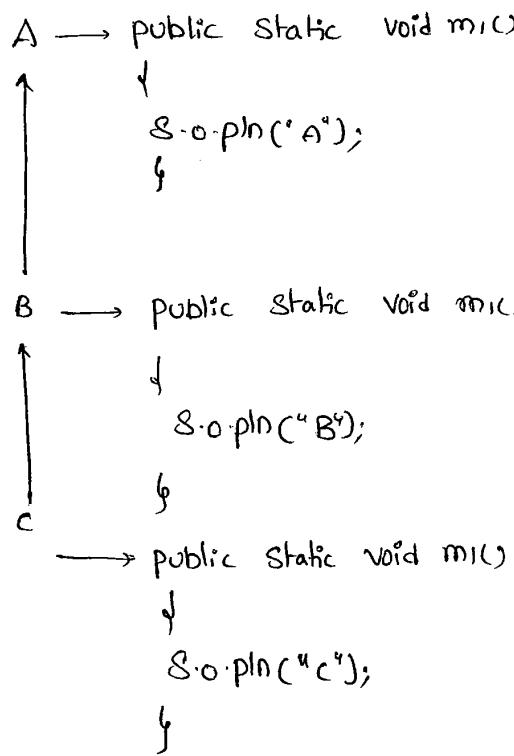
c.m₁(); → c ↗

((B)c).m₁(); → c ↗ → B b = new B();
b.m₁();

((A)c).m₁(); → c ↗

→ A a = new A();
a.m₁();

Eg 1:



→ C c = new C();

c.m1(); // C

→ ((B)c).m1(); // B

→ ((A)c).m1(); // A

Eg 2:

A → int x = 777;



B → int x = 888;



C → int x = 999;

C c = new C();

S.o.println(c.x); 999

S.o.println(((B)c).x); 888

S.o.println(((A)((B)c)).x); 777

(because The overriding Concept is not applicable
for variable).

→ If we declare all Variables as Static then There is no chance to change the O/p.

Note:-

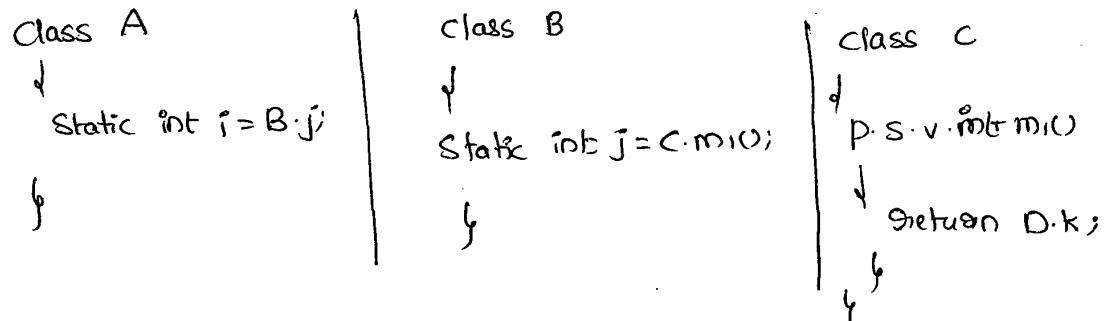
→ whether the Variable is Static or instance Variable Resolution should be done based on Reference type but not based on Runtime Object.

Coupling

Coupling :-

→ The degree of dependency b/w The Components is Called "Coupling"

Ex:-



Class 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 ^{remaining} any Component we can't modify any Component's

Hence, enhancement will become difficult.

→ It reduces maintainability.

→ It doesn't promote reusability.

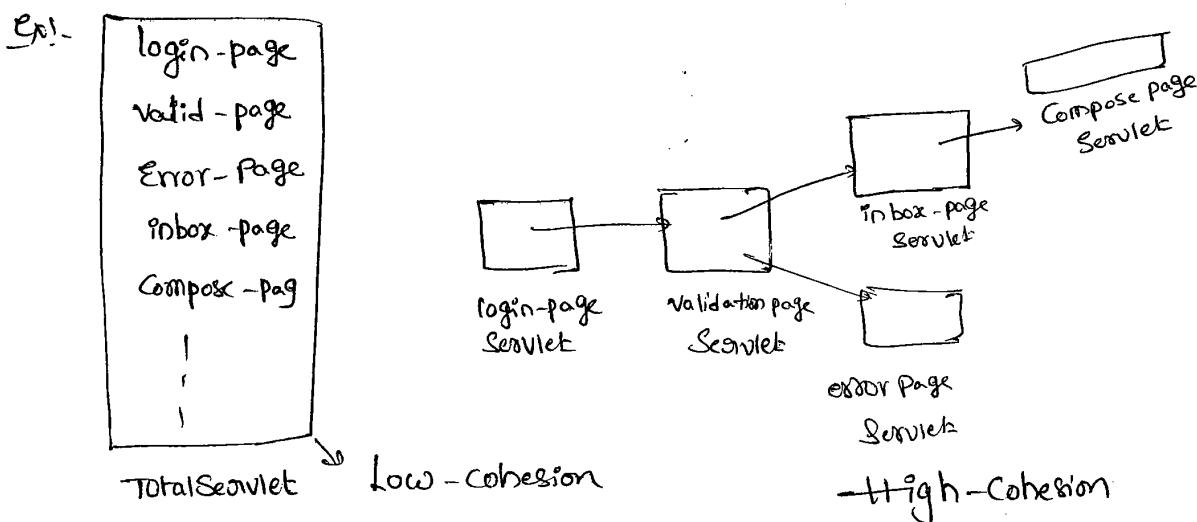
→ Hence it is highly recommended to maintain loosely coupling & dependency b/w the components should be as less as possible.

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



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

- (1) without affecting remaining components we can modify any component hence enhancement will become very easy

- (i) SE improves maintainability of the application
- (ii) SE 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~~ one object is created then
call those values from the current class.

Kondalu-7@yahoo.com

25, 26

① Collection Framework (1 - 43)

Collection (6 - 24)

Map (25 - 37)

Collections class (38 - 46)

Arrays class (41 - 43)

② Generics (44 - 52)

③ Multithreading (53 - 76)

Synchronization (67 - 76)

④

Re

④ Regular Expressions (77 - 82)

⑤ Enumeration (83 - 89)

⑥ I18N (90 - 95)

⑦ Development (96 - 101)

20311

Collection framework

→ An Array is an indexed collection of fixed no. of homogeneous data elements.

* Limitations of object arrays :-

- 1) Arrays are fixed in size. i.e., once we created an array there is no chance of increasing or decreasing size based on our requirement.
- 2) Hence, to use arrays Concept Compulsory we should know the size in advance, which may not possible always.
- 3) Arrays can hold only Homogeneous data elements. i.e., (Same type)

Ex:-

```
Student[] S = new Student[1000];
```

```
S[0] = new Student[]; ✓
```

```
S[1] = new Student[]; ✓
```

```
S[2] = new Customer[]; ✗ cpt :- Incompatible types
```

↳ found: Customer

↳ required: Student.

↳ But we can resolve this problem by using Object-type arrays.

Ex:-

```
Object[] a = new Object[1000];
```

```
a[0] = new Student[]; ✓
```

```
a[1] = new Customer[]; ✓
```

(3) Arrays Concept not built based on some datastructure, Hence

→ Standard method support is not available, for every requirement.

→ Compulsory programmer is responsible to write the logic.

→ To resolve the above problems Sun people introduced Collections Concept.

→ Advantages of Collections over arrays :-

- (1) Collections are growable in nature. Hence based on our requirement we can increase or decrease the size.
- (2) Collections can hold both Homogeneous & Heterogeneous objects.
- (3) Every Collection class is implemented based on some data structures. Hence predefined method support is available for every requirement.

dis. of Collections :-

→ Performance point of view Collections are not recommended to use. This is the limitation of Collections.

Difference b/w arrays & Collections :-

Array	Collections (AL, VL, LL - ...)
1) Arrays are fixed in size	1) Collections are growable in nature
2) Memory point of view arrays	2) memory point of view Collections
• Concept is not recommended to use	Concept is highly recommended to use.
3) Performance point of view arrays Concept is highly recommended to use.	3) performance point of view Collections is not recommended to use.
4) Arrays can hold only homogeneous data elements.	4) Collections can hold both Homogeneous & Heterogeneous objects.
5) There is no underlying d.s for arrays. Hence predefined method support is not available	5) Underlying D.S is available for every Collection class. Hence predefined method support is available.

→ Arrays can be used to hold both primitives & objects.

→ Collections can be used to hold only objects but not for primitives.

Collection :-

→ A group of individual objects as a single entity is called Collection.

Collection framework :-

→ It defines several classes & interfaces, which can be used to represent a group of objects as a single Entity.

Terminology:-

Java	C++
Collection	Container
Collection framework	STL (Standard Template Library)

9 - Key interfaces of Collection framework :-

① Collection (Interface) :-

→ If we want to represent a group of individual objects as a single Entity then we should go for Collection.

→ In general Collection Interface is considered as root Interface of Collection framework.

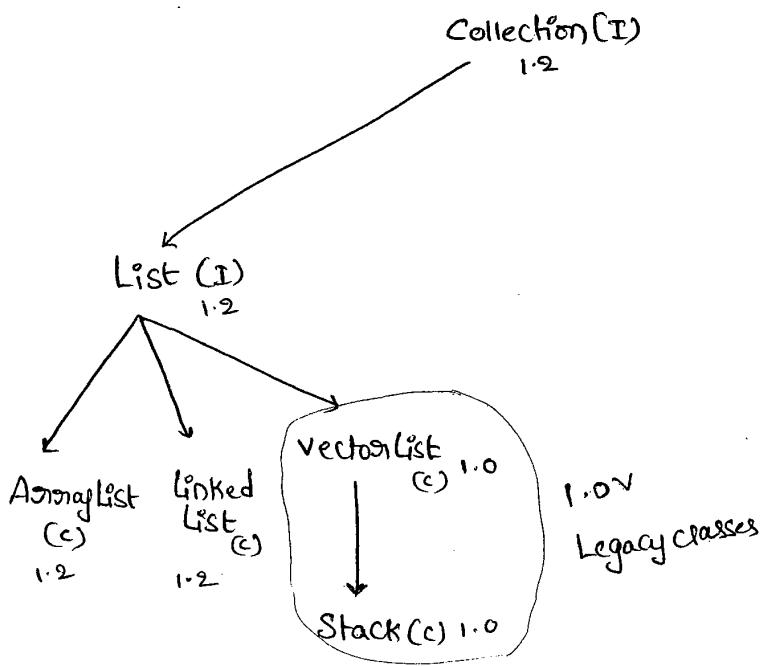
→ Collection Interface defines the most common methods which can be applicable for any Collection object.

Collection vs Collections :-

- Collection is an interface, Can be used to represent a group of individual object as a Single Entity, where as,
- Collections is an Utility class, present in java.util package, to define Several utility methods for Collections.

1) List (Interface) :-

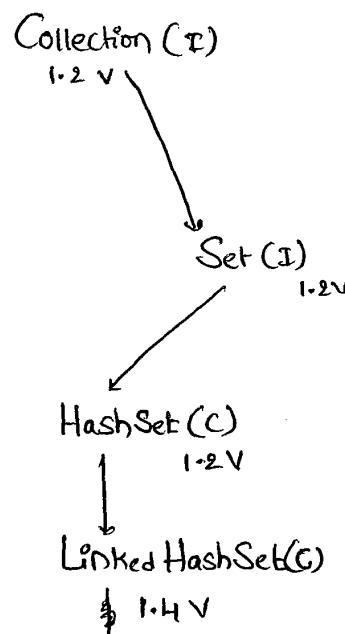
- It is the child Interface of Collection.
- If we want to represent a group of individual objects where insertion order is preserved & duplicates are allowed. Then we should go for List.



- Vector & Stack classes are re engineered in 1.2 version to fit into Collection framework.

③ Set (Interface):-

- It is the child interface of Collection.
- If we want to represent a group of individual objects where "duplicates are not allowed & insertion order is not preserved". Then we should go for "Set".

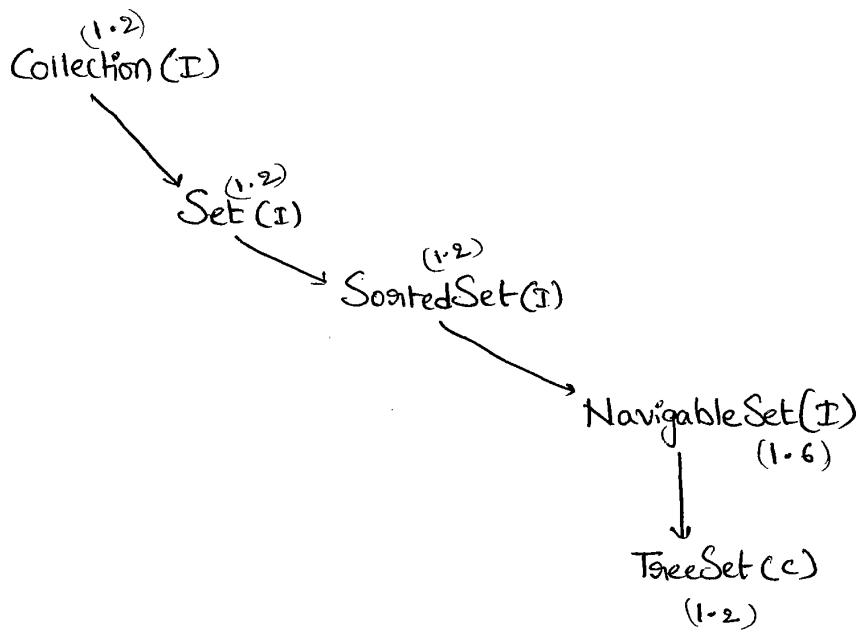


④ SortedSet (I):-

- It is the child interface of Set.
- if we want to represent a group of ~~unique~~^{individual} objects, according to some sorting order. Then we should go for SortedSet.

⑤ NavigableSet (I) :-

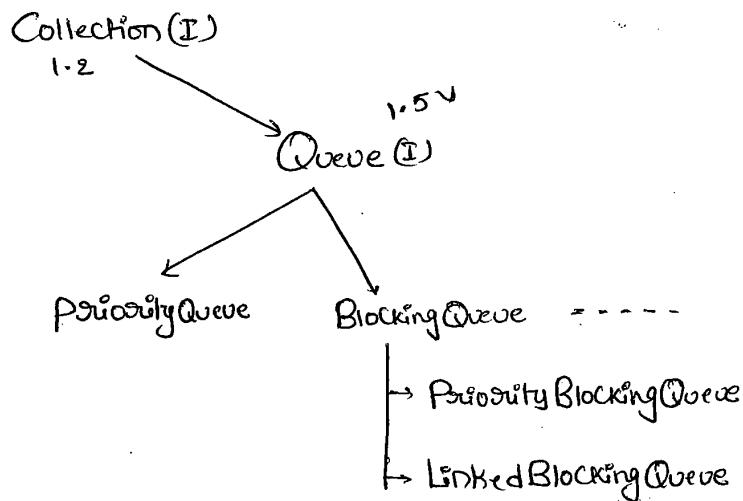
- It is the child interface of SortedSet, to provide several methods for Navigation purposes.
- It is introduced in 1.6 version.



⑥ Queue (I) :- (1.5v)

→ It is the child Interface of Collection.

→ If we want to represent a group of individual objects, prior to processing, Then we should go for Queue.

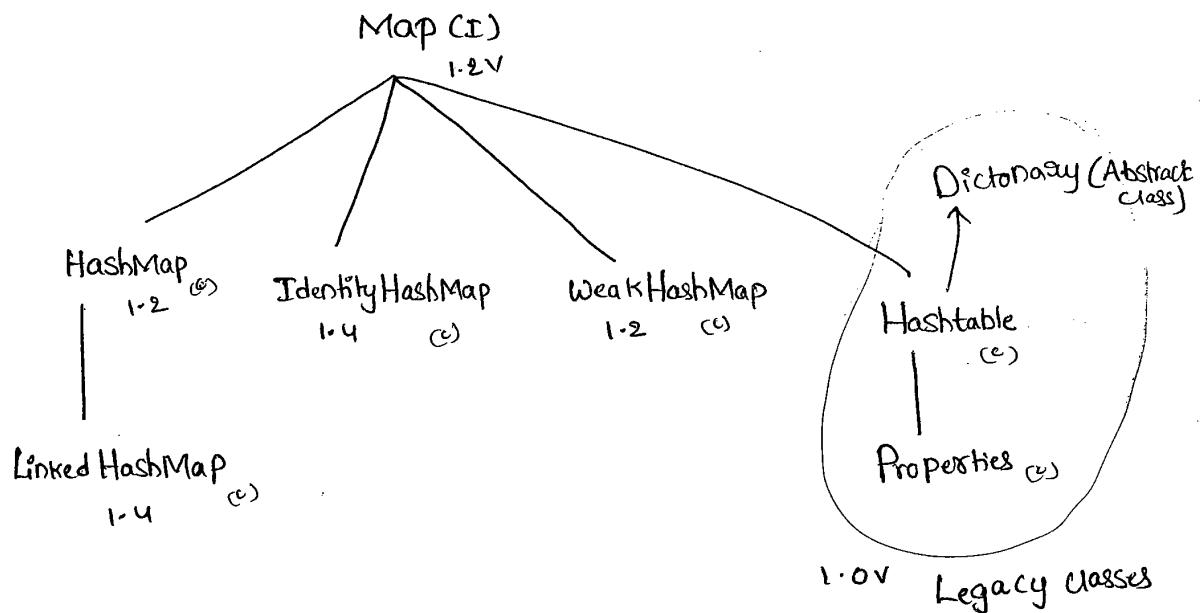


Note :-

- all the above Interfaces (Collection, List, Set, SortedSet, NavigableSet, Queue) meant for representing a group of individual objects.
- If we want to represent a group of objects as key-value pairs Then We should go for Map.

(7) Map(I) :-

- If we want to represent a group of objects as key-value pairs Then we should go for Map.
- Both Key & value are objects only.
- Duplicate keys are not allowed, But values can be duplicated.



Note:-

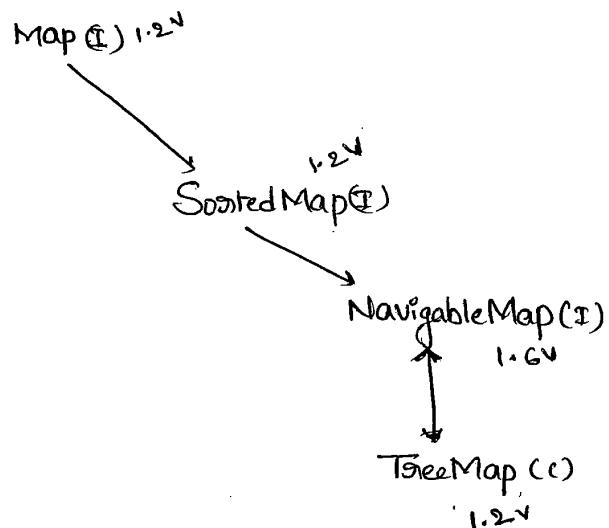
→ Map is not child interface of Collection.

(8) SortedMap (I) :-

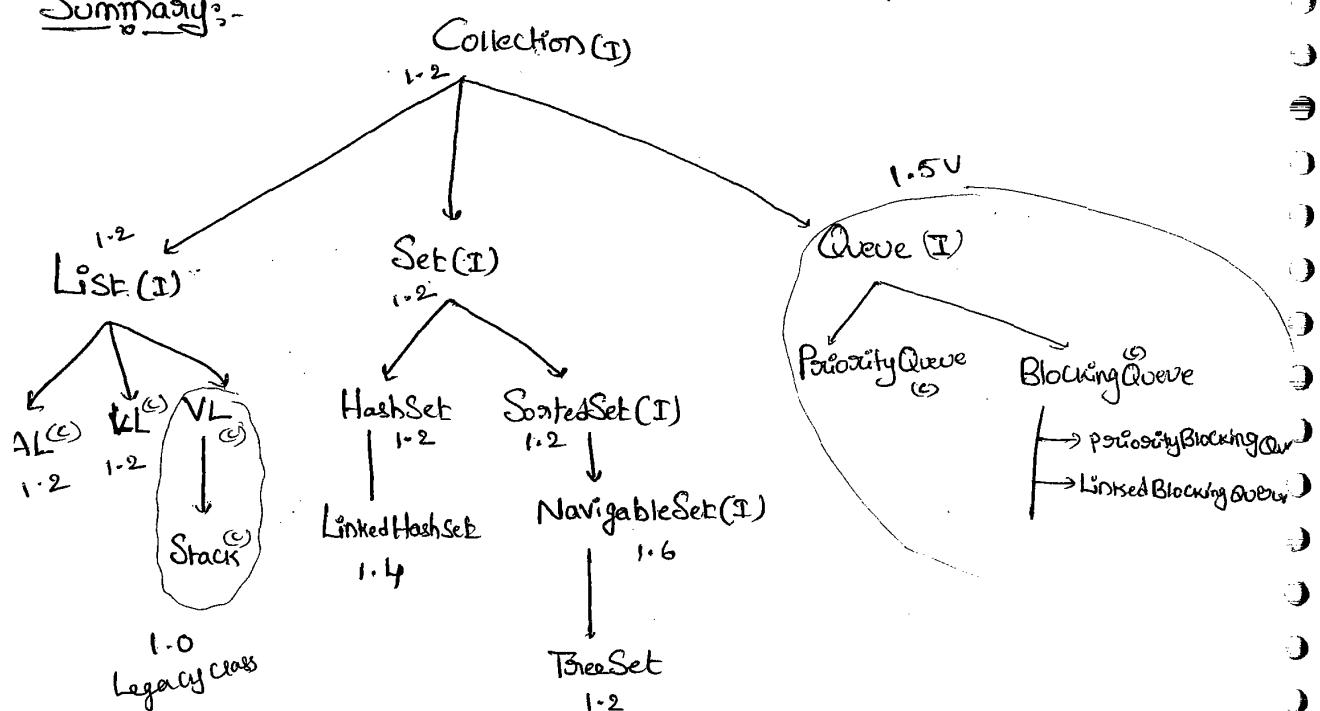
- If we want to represent a group of objects as key-value pairs according to some sorting order. Then we should go for SortedMap.
- Sorting should be done only based on Keys, but not based-on values.
- SortedMap is child interface of Map.

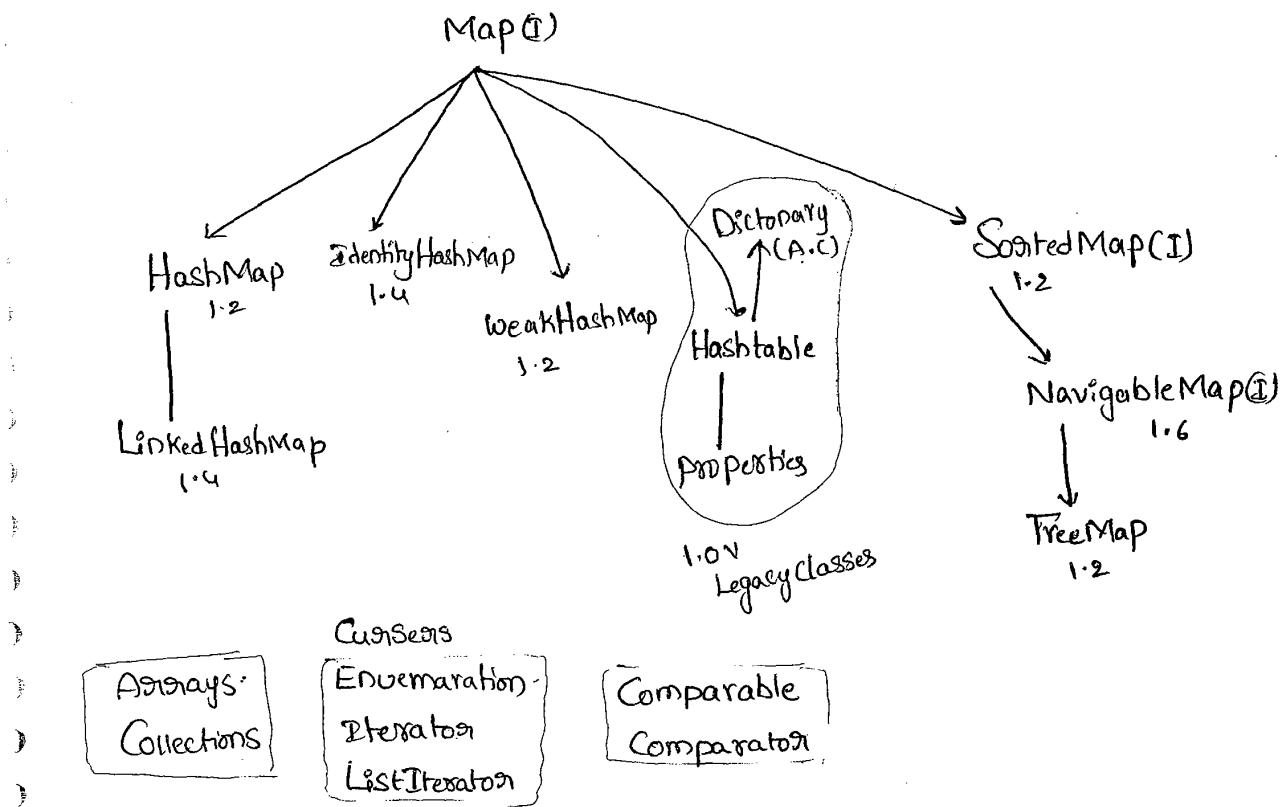
(9) NavigableMap (I)

→ It is the child Interface of SortedMap & define Several methods for Navigation purposes.



Summary:-





→ In the Collection framework the following are Legacy characters

- (1) Enumeration (I)
 - (2) Dictionary (A..C)
 - (3) Vector
 - (4) Stack
 - (5) Hashtable
 - (6) Properties
- classes } 1.0V

Collection framework :-

Collection (I) :-

- If we want to represent a group of individual objects as a single entity then we should go for Collection.
- Collection Interface defines the most common methods which can be applied for any Collection object.
- The following is the list of methods present in Collection Interface.

- ① boolean add(Object o)
- ② boolean addAll(Collection c)
- ③ boolean remove(Object o)
- ④ boolean removeAll(Collection c)
- ⑤ boolean retainAll(Collection c)
→ To remove all objects except those present in C.
- ⑥ void clear()
- ⑦ boolean isEmpty()
- ⑧ int size()
- ⑨ boolean contains(Object o)
- ⑩ boolean containsAll(Collection c)
- ⑪ Object[] toArray()
- ⑫ Iterator iterator()

② List (I) :-

→ List is the child Interface of Collection.

→ If we want to represent a group of individual objects where duplicate Objects are allowed & insertion Order is preserved. Then we Should go for List.

→ Insertion Order will be preserved by means of Index.

→ we Can differentiate duplicate objects by using Index. Hence Index plays a very important role in List.

→ List Interface defines the following methods.

- ① boolean add(int index, Object o)
- ② boolean addAll(int index, Collection c)
- ③ Object remove(int index)
- ④ Object get(int index)
- ⑤ Object ^{old} set(int index, Object new)
- ⑥ int indexOf(Object o)
- ⑦ int lastIndexOf(Object o)
- ⑧ ListIterator listIterator()

It Contains 4 Classes:-

(i) ArrayList (C) :-

(ii) LinkedList (C) :-

(iii) VectorList (C) :-

(iv) Stack (C) :-

(i) ArrayList (c) :-

- The underlying datastructure for ArrayList is Resizable Array or Growable Array.
- Insertion Order is preserved.
- Duplicate objects are allowed.
- Heterogeneous Objects are allowed.
- Null insertion is possible.

Constructors :-

① ArrayList Al = new ArrayList();

- Creates an Empty ArrayList Object, with default initial Capacity 10.
- Once AL reaches it's max capacity then a new AL object will be created with.

$$\text{New Capacity} = \text{Current Capacity} * \frac{3}{2} + 1$$

② ArrayList l = new ArrayList(int initialCapacity);

- Creates an Empty ArrayList Object with the Specified initial Capacity.

③ ArrayList l = new ArrayList(Collection c);

- Creates an Equivalent ArrayList object for the Given Collection object ie, This Constructor is for cloning b/w Collection objects

```

Ex) import java.util.*;

class ArrayListDemo
{
    P.S.V.m(String[] args)
    {
        ArrayList a = new ArrayList();
        a.add("A");
        a.add(10);
        a.add('A');
        a.add(null);
        System.out.println(a); // [A, 10, A, null]
        a.remove(2);
        System.out.println(a); // [A, 10, null]
        a.add(2, "M"); // [A, 10, M, null]
        a.add("N"); // [A, 10, M, null, N]
        System.out.println(a); // [A, 10, M, null, N]
    }
}

```

Note:-

In Every Collection class toString() is overridden to return

its Content directly in the following format.

[obj1, obj2, obj3, ...]

→ Usually we can use Collection to store & transfer Objects. to provide

Support for this requirement Every Collection class implements

Serializable & Clonable Interfaces.

- `ArrayList` & `Vector` classes implements Random Access Interface, So that any random element we can access with same speed. Hence, if our frequent operation is gettable operation then best suitable data structure is `ArrayList`. (Advantage)
- If our frequent operation is Insertion or deletion, ^{operation} in the middle then `ArrayList` is the worst choice, because it required several shift operations. (disadvantage)

differences b/w `ArrayList` & `Vector`

<u><code>ArrayList</code></u>	<u><code>Vector</code></u>
① No method is Synchronized	① Every method is Synchronized
② multiple threads can access <code>ArrayList</code> simultaneously, hence <code>ArrayList</code> object is not threadsafe	② At any point only one thread is allowed to operate on <code>Vector</code> Object at a time. Hence <code>Vector</code> Object is Thread Safe.
③ Threads are not required to wait, & hence performance is high.	③ It increases waiting time of threads & hence performance is low.
④ Introduced in 1.0 version & hence it is non-legacy	④ Introduced in 1.0 version & hence it is Legacy.

Q) How to get Synchronized Version of ArrayList?

- A) → By using Collections Class synchronizedList() we can get synchronized version of ArrayList.

Public static List synchronizedList(List l)

Ex:-

ArrayList l = new ArrayList();

List l₁ = Collections.synchronizedList(l)

↓
Synchronized

↓
Non-Synchronized,

→ Similarly we can get synchronized version of Set & Map objects by using the following methods respectively.

① Public static Set synchronizedSet(Set s)

② public static Map synchronizedMap(Map m)

Note:-

→ If our frequent operation is insertion or deletion in the middle then ArrayList is not recommended. To handle this requirement we should go for LinkedList.

③) **LinkedList** (C) :-

- The underlying datastructure is double LinkedList.
- Insertion order is preserved.
- Duplicate objects are allowed.
- Heterogeneous " "
- Null insertion is possible.
- Implements Serializable & Clonable interfaces but not RandomAccess-
interface.
- Best Suitable if our frequent operation insertion or deletion
in the middle.
- Worst choice if our frequent operation is retrieval.

Constructors:-

- ① `LinkedList l = new LinkedList();`
→ Creates an Empty LinkedList object.
- ② `LinkedList l = new LinkedList(Collection c)`
→ for interConversion b/w Collection objects.

LinkedList Specific methods:-

- Usually we can use LinkedList to implements Stacks & Queues
to support this requirements LinkedList class define the following
Six Specific methods.

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

Ex:-

```

import java.util.*;
class LinkedListDemo
{
    public static void main (String [] args)
    {
        LinkedList l = new LinkedList();
        l.add("durga");
        l.add(30);
        l.add(null);
        [durga, 30, null, durga], l.add("durga"),      ccc venkey durga 30 null durga
        l.set (0, "Software");
        [venkey, Software, 30, null]
        l.add (0, "venkey");
        [venkey, Software, 30, null]
        l.removeLast();
        [ccc, venkey, Software, 30, null]
        l.addFirst ("ccc");
        S.o.println(l);
    }
}

```

Vector :-

- The underlying datastructure is Resizable array or growable array.
- Insertion Order is Preserved.
- duplicate objects are allowed.
- null insertion is possible
- Heterogeneous Objects are allowed.
- implements Serializable, Clonable & RandomAccess interfaces.
- Best Suitable if our frequent operation is Retrieval & worst choice if our frequent operation is insertion or deletion in the middle.
- Every method in vector is Synchronized. Hence vector object is ThreadSafe.

Constructors:-

(1) Vector v = new Vector();

→ Creates an Empty Vector object with default initial Capacity 10.

→ Once Vector reaches it's Max. Capacity a new Vector object will be created with double Capacity.

$$\text{New Capacity} = 2 * \text{Current Capacity}.$$

(2) Vector v = new Vector(int initialCapacity);

(3) Vector v = new Vector(int initialCapacity, int incrementalCapacity);

(4) Vector v = new Vector(Collection c);

Vector Specific methods :-

→ To add objects

① add(Object o) → C

② add(int index, Object o) → L

③ addElement(Object obj) → V

→ To remove Elements or objects

① remove(Object o) → C

② removeElement (Object o) → V

→ ③ remove (int index) → L

④ removeElementAt (int index) → V

⑤ clear () → C

⑥ removeAllElements () → V

→ To retrieve elements

① get (int index) → L

② elementAt (int index) → Y

③ firstElement (); → Y

④ lastElement (); → V

→ Other methods

① int size();

② int capacity();

* ③ Enumeration elements();

```

Eg:- import java.util.*;

class Demo1
{
    public static void main(String[] args)
    {
        Vector v = new Vector();
        System.out.println(v.capacity());
        for(int i=1; i<=10; i++)
        {
            v.addElement(i);
        }
        System.out.println(v.capacity());
        v.addElement("A");
        System.out.println(v.capacity());
        System.out.println(v);
    }
}

```

Op:-

10
10
20

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, A]

v.size() // no. of objects
 // [1] object
 v.removeElement(9) // [1, 2, 3, 4, 5, 6, 7, 8, 10, A]
 v.removeElementAt(3) // [1, 2, 3, 5, 6, 7, 8, 10, A]
 v.removeAllElements() // []

(ii) Stack (c) :- (LIFO)

→ It is the child class of vector Contains only one Constructor

(i) Stack s = new Stack();

Methods:-

(i) Object push(Object o)

To Insert an object into the Stack

(ii) Object pop();

To Remove and returns top of stack.

(iii) Object peek();

To return top of the stack.

(iv) boolean empty();

Returns true when Stack is Empty.

(v) int search(Object o)

Returns the offset from top of the stack if the object

is available, otherwise returns -1.

Ex:- import java.util.*;

Class StackDemo

{ public static void main(String[] args)

{ Stack s = new Stack();

s.push("A");

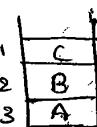
s.push("B");

s.push("C");

s.print(); // [A B C]

s.print(s.search("A")); s

s.print(s.search("Z")); -1

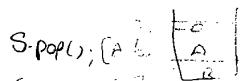


offset

s.search("A"); 3

s.search("C"); 1

s.search("Z"); -1



offset

s.pop(); [B C]

top: [C]

size: 2

Cursors :

Types of Cursors :

→ If we want to get objects one by one from the Collection we should go for Cursor.

→ There are 3 types of Cursors available in Java.

(i) Enumeration (1.0v)

(ii) Iterator (1.2v)

(iii) ListIterator (1.2v)

(i) Enumeration (1.0v)

→ It is a Cursor to retrieve Objects one by one from the Collection.

→ It is applicable for legacy classes.

→ We can Create Enumeration object by using elements()

```
public Enumeration elements();
```

e.g:-

```
Enumeration e = v.elements();
```

vector object

→ Enumeration Interface defines the following 2 methods.

(i) public boolean hasMoreElements();

(ii) public Object nextElement();

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

```

class EnumerationDemo
{
    public static void main(String[] args)
    {
        Vector v = new Vector();
        for(int i=0; i<=10; i++)
        {
            v.addElement(i);
        }
        System.out.println(v);  [0, 1, 2, 3, ..., 10]
        Enumeration e = v.elements();
        while(e.hasMoreElements())
        {
            Integer I = (Integer)e.nextElement();
            if(I%2 == 0)
                System.out.println(I);
        }
        System.out.println(v);  [0, 1, 2, 3, 4, ..., 10]
    }
}

O/P:- [0, 1, 2, 3, ..., 10]
0
2
4
6
8
10
[0, 1, 2, 3, ..., 10]

```

Limitations of Enumeration :-

- Enumeration Concept is applicable only for Legacy Classes & hence it is not a universal Cursor.
- By using Enumeration we can get only ReadAccess & we can't perform any remove operations.
- To over Come These Limitations SUN people introduced Iterator in 1.2 Version.

Iterator :-

- We can apply Iterator Concept for any Collection object. It is a universal Cursor.
- While Iterating we can perform remove operation also, in addition to read operation.
- We can get Iterator object by iterator() of Collection Interface.

Iterator it = C.iterator()

Any Collection object

- Iterator Interface defines The following 3 methods.

- public boolean hasNext();
- public Object next();
- public void remove();

Eg:-

```
import java.util.*;
```

```
class HashSetDemo
```

```
{
```

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

```
{
```

```
    HashSet h = new HashSet();
```

```
    h.add("B");
```

```
    h.add("C");
```

```
    h.add("D");
```

```
    h.add("Z");
```

```
    h.add(null);
```

```
    h.add(10);
```

```
    System.out.println(h.add("Z")); // false
```

```
    System.out.println(h); // [null, D, B, C, 10, Z]
```

```
}
```

O/P:-

false

[null, D, B, C, 10, Z]

Note:- Insertion order is not preserved

(ii) LinkedHashSet :-

→ LinkedHashSet is the child class of HashSet.

→ It is exactly same as HashSet except the following differences.

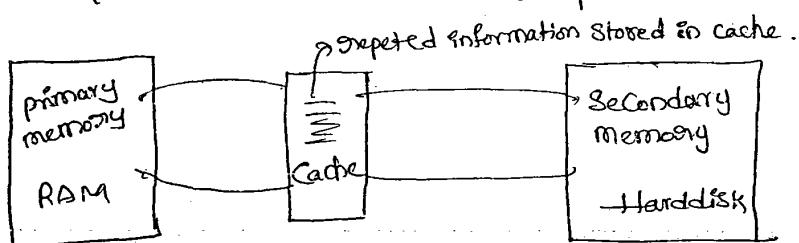
④ HashSet	LinkedHashSet
i) The underlying D.S is HashTable	i) The underlying D.S is a Combination of HashTable & Linked List
ii) Insertion order is not preserved	ii) Insertion order is preserved.
iii) Introduced in 1.2v	iii) Introduced in 1.4v

→ In the above program if we are replacing HashSet with
LinkedHashSet the following is the O/P.

O/P :- [B, C, D, Z, null, 10] i.e., insertion order is preserved.

Note :-

→ The main important application area of LinkedHashSet & LinkedHashMap is implementing Cache applications. where duplicates are not allowed & insertion order must be preserved.



ii) SortedSet (I) :-

→ SE is the Child Interface of Set.

→ If we want to represent a group of individual Objects according to some Sorting order. Then we should go for SortedSet

→ SortedSet Interface defines the following 6 Specific methods

(i) Object first()

→ Returns the first element of SortedSet.

(ii) Object last()

→ Returns last element of SortedSet

(iii) SortedSet headSet(Object obj)

→ Returns the SortedSet whose elements are less than obj.

(iv) SortedSet tailSet (Object obj)

17
145

\geq

→ Returns the SortedSet whose elements are greater than or equal to obj

(v) SortedSet subSet (Object obj1, Object obj2)

→ Returns the SortedSet whose elements are $\geq obj1$ but $< obj2$

(vi) Comparator Comparator()

→ Returns Comparator object describes underlying Sorting technique

→ If we used default natural Sorting order Then we will get null.

Ex:-

100
101
103
104
107
109

- ① first() → 100
- ② last() → 109
- ③ headSet(104) → [100, 101, 103]
- ④ tailSet(104) → [104, 107, 109]
- ⑤ subSet(101, 107) → [101, 103, 104]
- ⑥ Comparator() → null

Note:-

→ The default Natural Sorting order for the no.'s is ascending order

→ The default Natural Sorting order for Characters & Strings are is Alphabetical order (Dictionary based Order).

TreeSet (c) :-

- The underlying data structure is Balanced Tree.
- duplicate objects are not allowed.
- Insertion order is not preserved. because Objects will be inserted according to Some Sorting order.
- Heterogeneous objects are not allowed. otherwise We will get "ClassCastException". & null insertion is not possible. for IMP(Empty Set).

Constructors:-

(i) TreeSet t = new TreeSet();

- Creates an Empty TreeSet object where the Sorting order is default Natural Sorting order.

(ii) TreeSet t = new TreeSet(Comparator c)

- Creates an Empty TreeSet object where the Sorting order is Customized Sorting order Specified by Comparator object.

(iii) TreeSet t = new TreeSet(Collection c)

(iv) TreeSet t = new TreeSet(SortedSet c)

Ex:- import java.util.*;

```
class TreeSetDemo  
{
```

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

```
}
```

18
1496

```
TreeSet t = new TreeSet();
t.add("A");
t.add("a");
t.add("B");
t.add("z");
t.add("L");
//t.add(new Integer(10)); //CCE ClassCastException
//t.add(null); //→ NPE
System.out.println(t); [A, B, z, L, a]
```

→ Null acceptance :-

- (i) For the Non-Empty TreeSet if we are trying to insert null we will get Nullpointer Exception (NPE).
 - (ii) For the Empty TreeSet add the first element null insertion is always possible.
 - (iii) But after inserting that null, if we are trying to insert any other, we will get NullpointerException (NPE).

```

    eg:- import java.util.*;
    class TreeSetDemo1
    {
        public static void main (String [] args)
        {
            TreeSet t = new TreeSet();
            t.add (new StringBuffer("A"));
            t.add (new StringBuffer("Z"));
            t.add (new StringBuffer("Y"));
            t.add (new StringBuffer("B"));
            System.out.println(t);
        }
    }

```

- If we are depending on default natural sorting order Compulsory objects should be homogeneous & Comparable otherwise we will get ClassCastException (CCE)
- An object is Said to be Comparable iff The Corresponding class implements Comparable Interface.
- String class & all wrapper classes already implements Comparable Interface whereas StringBuffer doesn't implement Comparable Interface. Hence, In the above Example we got ClassCastException.

Comparable Interface :-

- This Interface present in java.lang package & Contains only one method "compareTo".

public int compareTo(Object obj)

Obj1.compareTo(Obj2)

- Returns -ve iff obj1 has to Come before obj2.
- Returns +ve iff obj2 has to Come after obj2.
- Returns 0 iff obj1 & obj2 are equal (duplicate)

e.g.: `import java.util.*;`

`class Test`

`{`

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

`{ S.o.println("A".compareTo("z")); } // -ve -25`

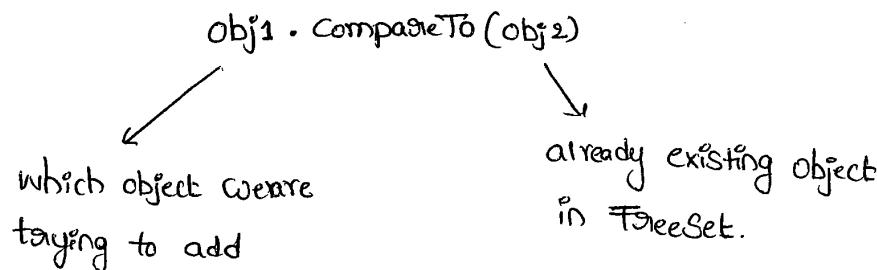
`S.o.println("z".compareTo("k")); } // +ve +5`

`S.o.println("A".compareTo("A")); } // 0`

→ When we're add depending on default Natural Sorting order

internally JVM calls `com ObjectTo()`.

→ Based on the return-type JVM identifies the location of the element in sorting order.



-) → Returns -ve iff obj1 has to come before obj2.
-) → Returns +ve iff obj1 has to come after obj2.
-) → Returns 0 iff obj1 & obj2 are equal

Eg:-

`TreeSet t = new TreeSet();`

`t.add("z");`

`t.add("k");` → "k".`Comparable(z)`; -ve

`t.add("D");` → "D".`Comparable(k)`; -ve

`t.add("M");` → "M".`Comparable(D)` ⇒ +ve

`t.add("O");` → "M".`Comparable(k)` ⇒ +ve

"M".`Comparable(z)`; → -ve

"D".`Comparable(O)` ⇒ 0

`S.out(t);`

`[O ,K,M,z]`

ClassCastException: NPE

`null . Comparable("O")` ⇒ RE ⇒ NPE

→ If we are not satisfied with default Natural Sorting order
③ if the ^{default} Natural Sorting order is not already available. Then
we can define our own Customized Sorting by using Comparator

- * Comparable method for default Natural Sorting order.
- * Comparator method for Customized Sorting order.

Comparator (I) :-

→ Comparator Interface present in `java.util` package & defines
the following 2 methods.

① `public int compare(Object obj1, Object obj2);`

- returns -ve iff obj1 has to come before obj2
- returns +ve iff obj1 has to come after obj2
- returns 0 iff obj1 & obj2 are equal (duplicate).

obj1 ⇒ which object we are trying to add

obj2 ⇒ already existing object

② `public boolean equals(Object obj)`

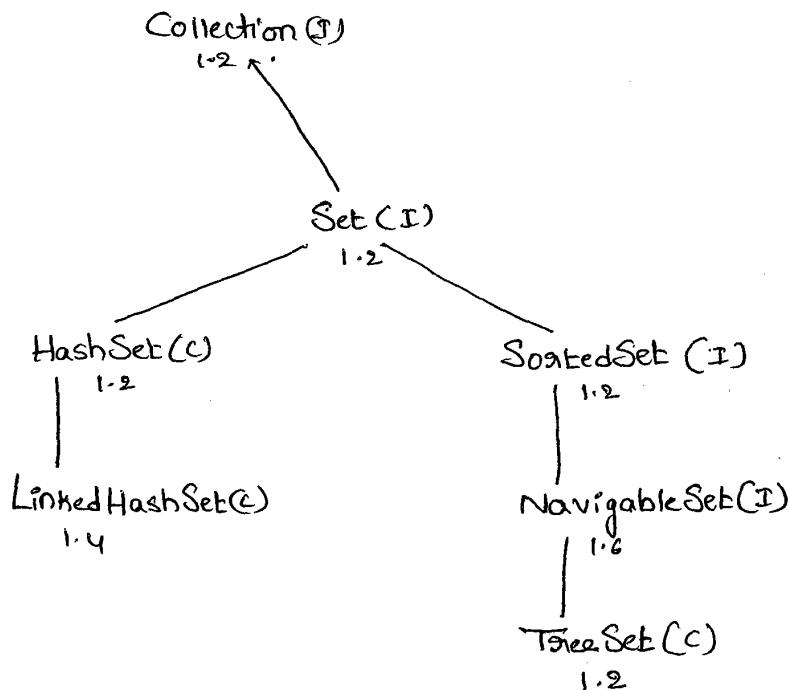
→ whenever we are implementing Comparator Interface Compulsory
we should provide implementation for `compare()`, 2nd method

• `equals()` implementation is optional, because it is already available for
our class from Object class through inheritance.

② Set (I) :-

118

- Set is child Interface of Collection.
- If we want to represent a group of objects where duplicates are not allowed & insertion order is not preserved, then we should go for Set.



- Set Interface does not contain any method we have to use Only Collection Interface method.
- (i) HashSet (C) :-
- The underlying datastructure is HashTable.
- Duplicate objects are not allowed.
- If we are trying to add duplicate objects, we won't to get any C.E or R.E. Add() simply returns false, to hashCode of the object
- Insertion order is not preserved & all objects are inserted according

- Heterogeneous Objects are allowed.
- Null insertion is possible (only once) because duplicates are not allowed.
- HashSet Implements Serializable & Clonable Interfaces.

* Constructors:-

- 1) HashSet h = new HashSet();
→ Creates an Empty HashSet object with default default initial Capacity 16 & default fillRatio 0.75 (75%).
- 2) HashSet h = new HashSet(int initialCapacity);
→ Creates an Empty HashSet object with the specified initial Capacity & default fillRatio is 0.75.
- 3) HashSet h = new HashSet(int initialCapacity, float fillratio);
→ 0 to 1
- 4) HashSet h = new HashSet(Collection c);

Fillratio:-

- After Completing, The Specified ratio Then only a new HashSet Object will be Created That particular ratio is called fillratio or load factor.
- The default fillratio is 0.75 but we can customized this value.

```

Eg:- import java.util.*;
class IteratorDemo
{
    public static void main(String[] args)
    {
        ArrayList l = new ArrayList();
        for(int i=0; i<=10; i++)
        {
            l.add(i);
        }
        System.out.println(l); [0, 1, 2, 3, ..., 10]
        Iterator it = l.iterator();
        while(it.hasNext())
        {
            Integer I = (Integer)it.next();
            if(I%2 == 0)
            {
                System.out.println(I); 2
                System.out.println(I); 4
                System.out.println(I); 6
                System.out.println(I); 8
                System.out.println(I); 10
            }
            else
            {
                it.remove();
            }
        }
        System.out.println(l); [0, 2, 4, 6, 8, 10]
    }
}

```

Limitations of Iterator :-

- (i) In the Case of Iterator & Enumeration we can always move towards the forward direction & we can't move backward direction.
i.e These Cursors are Single directional Cursors but not Bidirectional.
- (ii) While performing Iteration we can perform only ~~read & remove~~ operations

We Can't perform Replacement & Addition of New Objects.

→ To Resolve These problem Sun people Introduced ListIterator
in 1.2 Version.

3) List Iterator :-

→ ListIterator is the child Interface of Iterator.

→ While Iterating Objects by ListIterator we can move either to
the forward or to the Backward direction. i.e ListIterator
is a Bidirectional Cursor.

→ While Iterating By ListIterator we can perform replacement &
addition of new objects also in addition to Read & Remove Operations.

→ We can Create ListIterator object by using ListIterator(~~List~~)
List Interface.

any List object
↓ ↓
ListIterator litor = l.ListIterator();

→ ListIterator Interface defines the following 9 methods.

Forward }
(i) public boolean hasNext();
(ii) public Object next();
(iii) public int nextIndex();

Backward }
(iv) public boolean hasPrevious();
(v) public Object previous();
(vi) public int previousIndex();

- ⑦ public void remove();
- ⑧ public void set(Object new); → replace an object with new object
- ⑨ public void add(Object new); → add new obj.

Eg:-

```
import java.util.*;
class ListIteratorDemo
{
    Public static void main(String[] args)
    {
```

```
    LinkedList l = new LinkedList();
```

```
    l.add("balakrishna");
    l.add("venky");
    l.add("chaitu");
    l.add("nag");
```

```
S.o.println(l); [ balakrishna , venky , chaitu , nag ]
```

```
LinkedList
ListIterator ltr = l.listIterator();
```

```
while (ltr.hasNext())
{
```

```
    String s = (String) ltr.next();
```

```
    if (s.equals("venki"))
```

```
    {
        ltr.remove();
    }
```

```
    if (s.equals("chaitu"))
```

```
    {
        ltr.set("charan");
    }
```

```
    if (s.equals("nag"))
```

```
    {
        ltr.add("chaithu");
    }
```

```
} S.o.println(l);
```

[Balakrishna , charan , nag , Chaitu]

Note:-

→ Among 3 Cursors ListIterator is the most powerful Cursor,
But it is applicable only for List objects.

Comparison table of 3-Cursors :-

Property	Enumeration (1.0v)	Iterator (1.1v)	ListIterator (1.2v)
① Is it legacy	yes	No	No
② It is applicable only for Legacy classes		for any Collection objects	Only for List objects
③ Movement	Single direction (only forward)	Single direction (forward)	bi-directional (forward & backward)
④ How to get it?	By using elements() - method	By using iterator()	By using ListIterator()
⑤ Accessibility	only read	read & remove	read/remove/ replace/add
⑥ method	hasMoreElements() NextElement()	hasNext() next() remove()	9 methods

Eg:- import java.util.*;

```
class TreeSetDemo3
{
```

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

```
        Integer I1 = (Integer) obj1;
```

```
        Integer I2 = (Integer)
```

```
        TreeSet t = new TreeSet(new myComparator()); → ①
```

```
t.add(20);
```

```
t.add(0); → Compare(0, 20) → +ve
```

```
t.add(15); → Compare(15, 20) → +ve  
→ Compare(15, 0) → -ve
```

```
t.add(5); → Compare(5, 20) → +ve
```

```
t.add(10); → Compare(5, 0) → +ve  
→ Compare(5, 15) → -ve
```

```
s.print(t);
```

```
→ Compare(10, 20) → +ve  
→ Compare(10, 0) → +ve  
→ Compare(10, 15) → +ve  
→ Compare(10, 5) → -ve
```

```
[20, 15, 10, 5, 0]
```

```
class MyComparator implements Comparator
```

```
{
```

```
    public int compare(Object obj1, Object obj2)
```

```
{
```

```
        Integer I1 = (Integer) obj1;
```

```
        Integer I2 = (Integer) obj2;
```

```
        if(I1 < I2)
```

```
            return +100;
```

```
        else if(I1 > I2)
```

```
            return -100;
```

```
        else
```

```
            return 0;
```

```
} { return((I1 < I2) ? +1 : (I1 > I2) ? -1 : 0); }
```

- If we are not passing Comparator object at line 1 ①
 Then JVM internally calls compareTo() which is meant for default natural sorting order. In this case the o/p is [0, 5, 10, 15, 20].
- If we are passing comparator object at ① Then our own compare method will be executed which is meant for customized sorting order. These case the o/p is [20, 15, 10, 5, 0]

Various alternatives of implementing compare():-

```

class MyComparator implements Comparator
{
    public int compare (Object obj1, Object obj2)
    {
        Integer I1 = (Integer) obj1;
        Integer I2 = (Integer) obj2;

        // return I1.compareTo(I2) ;  ⇒ [0, 5, 10, 15, 20]
        // return -I1.compareTo(I2) ;  ⇒ [20, 15, 10, 5, 0]
        // return I2.compareTo(I1) ;  ⇒ [20, 15, 10, 5, 0]
        // return -I2.compareTo(I1) ;  ⇒ [0, 5, 10, 15, 20]

        // return -1 ;  ⇒ [10, 5, 15, 0, 20] ⇒ Reverse of insertion order
        // return +1 ;  ⇒ [20, 0, 15, 5, 10] ⇒ insertion order.

        // return 0 ;  ⇒ [20]
    }
}
  
```

* WAP to insert String Objects into the TreeSet where the Sorting order is reverse of alphabetical order.

```

④ import java.util.*;
class TreeSetDemo2
{
    public static void m(String[] args)
    {
        TreeSet t = new TreeSet(new myComparator());
        t.add("A");
        t.add("Z");
        t.add("K");
        t.add("B");
        t.add("a");
        System.out.println(t);
    }
}

```

Class MyComparator implements Comparator

```

{
    public int compare(Object obj1, Object obj2)
    {
        String s1 = (String) obj1;
        String s2 = obj2.toString(); ✓
        return -s1.compareTo(s2);
    }
}

```

Note:-

~~An object class Comparator
method doesn't contain strings
only contain object type so
objects can be converted into
strings by using toString~~

Note:-

→ In Objects & StringBuffer there is no Comparator, so we can convert into String.

* W.a.p to insert String & StringBuffer objects into the TreeSet
where the sorting order increasing length order. If two objects
having the same length then consider their alphabetical order

(*) import java.util.*;

Class TreeSetDemo12

```
{ public static void main(String[] args)
{
    TreeSet t = new TreeSet(new MyComparator());
    t.add("A");
    t.add(new StringBuffer("ABC"));
    t.add(new StringBuffer("AA"));
    t.add("XX");
    t.add("ABCD");
    t.add("A");
    System.out.println(t); [A, AA, XX, ABC, ABCD]
}}
```

class MyComparator implements Comparator

```
{ public int compare(Object obj1, Object obj2)
{}
```

String s₁ = obj1.toString();

String s₂ = obj2.toString();

int l₁ = s₁.length();

int l₂ = s₂.length();

if (l₁ < l₂)

return -1;

else (l₂ > l₁)

return +1;

else

return s₁.compareTo(s₂);

- * W.A.P TO insert StringBuffer objects into The TreeSet where the Sorting order alphabetical order?

```


import java.util.*;
class TreeSetDemo10
{
    public static void main(String[] args)
    {
        TreeSet t = new TreeSet(new MyComparator());
        t.add(new StringBuffer("A"));
        t.add(new StringBuffer("z"));
        t.add(new StringBuffer("K"));
        t.add(new StringBuffer("L"));
        System.out.println(t); // [A, K, L, z]
    }
}

class MyComparator implements Comparator
{
    public int compare(Object obj1, Object obj2)
    {
        String s1 = obj1.toString();
        String s2 = obj2.toString();
        return s1.compareTo(s2);
    }
}


```

O/P:- [A , K , L , Z]

So, SB Can be Convert into String

Note:-

→ In StringBuffer There is no CompareTo method

- If we are depending on default natural Sorting order Compulsory
- Objects should be Homogeneous & Comparable, otherwise we will get CCE
- If we are depending on our own Sorting by Comparator The Objects need not be Comparable & Homogeneous.

Comparable Vs Comparator :-

- ① For predefined Comparable classes default natural Sorting order is already available if we are not satisfied with that we can define our own Customized Sorting by using Comparator.

Ex:- String.

- ② For predefined Non-Comparable classes Default Natural Sorting order is not available Compulsory we should define Sorting by using Comparator object only.

Ex:- StringTokenizer.

- ③ For our own Customized classes to define default natural Sorting order we can go for Comparable & to define Customized Sorting we should go for Comparator.

Ex:- Employee, Student, Customer.

```
import java.util.*;  
  
class Employee implements Comparable  
{  
    int eid;  
  
    Employee(int eid)  
    {  
        this.eid = eid;  
    }  
  
    public String toString()  
    {  
        return "E-" + eid;  
    }  
  
    public int compareTo(Object obj)  
    {  
        int eid1 = this.eid;  
  
        Employee e2 = (Employee) obj;  
        int eid2 = this.e2.eid;  
  
        if (eid1 < eid2)  
            return -1;  
        else if (eid1 > eid2)  
            return +1;  
        else  
            return 0;  
    }  
  
    class CompDemo  
    {  
        public static void main(String[] args)  
    }
```

```
Employee e1 = new Employee(200);
```

```
Employee e2 = new Employee(100);
```

```
Employee e3 = new Employee(500);
```

```
Employee e4 = new Employee(300);
```

```
Employee e5 = new Employee(700);
```

```
TreeSet t1 = new TreeSet();
```

```
t1.add(e1);
```

```
t1.add(e2);
```

```
t1.add(e3);
```

```
t1.add(e4);
```

```
t1.add(e5);
```

```
s.o.println(t1); [E-100, E-200, E-500, E-700]
```

```
TreeSet t2 = new TreeSet(new MyComparator());
```

```
t2.add(e1);
```

```
t2.add(e2);
```

```
t2.add(e3);
```

```
t2.add(e4);
```

```
t2.add(e5);
```

```
s.o.println(t2); [E-700, E-500, E-200, E-100]
```

}

```
Class MyComparator implements Comparator
```

```
{
```

```
    public int compare(Object obj1, Object obj2)
```

```
{
```

```
        Employee e1 = (Employee) obj1;
```

```
        Employee e2 = (Employee) obj2;
```

```
} } return e2.compareTo(e1); // return -e1.compareTo(e2);
```

Q) Write a program to insert Employee objects into the TreeSet where default natural sorting order is ascending order of Salaries. If two Emp having the same salary then consider alphabetical orders of their names. &

* Write a Comparator class to define customized sorting which is alphabetical order of Employee names. If two Employees having the same name then consider descending order of their age.

* Comparison b/w Comparable & Comparator :-

Comparable	Comparator
<ul style="list-style-type: none"> 1) We can use Comparable to define default natural sorting order. 2) This interface present in Java.lang package. 3) defines only one method i.e compareTo() 4) All wrapper classes & String class implements Comparable interface 	<ul style="list-style-type: none"> 1) We can use Comparator to define customized sorting order. 2) This interface present in java.util package. 3) defines Two methods <ul style="list-style-type: none"> (i) compare() (ii) equals() 4) No predefined class implements Comparator Interface.

Comparison table for Set implemented classes?

Property	HashSet	LinkedHashSet	TreeSet
Underlying D.S	Hashtable	Hashtable + Linked List	Balanced Tree
1) Insertion Order	Not preserved	preserved	Not preserved
2) Sorting Order	N. A	N. A	preserved
3) Heterogeneous Objects	allowed	allowed	Not allowed
4) Duplicate Objects	not allowed	not allowed	not allowed
5) Null Acceptance	allowed (+)	allowed (-)	for the empty TreeSet add the first element Null insertion is possible, in all other cases we will get <u>NPE</u>

Map (I) :-

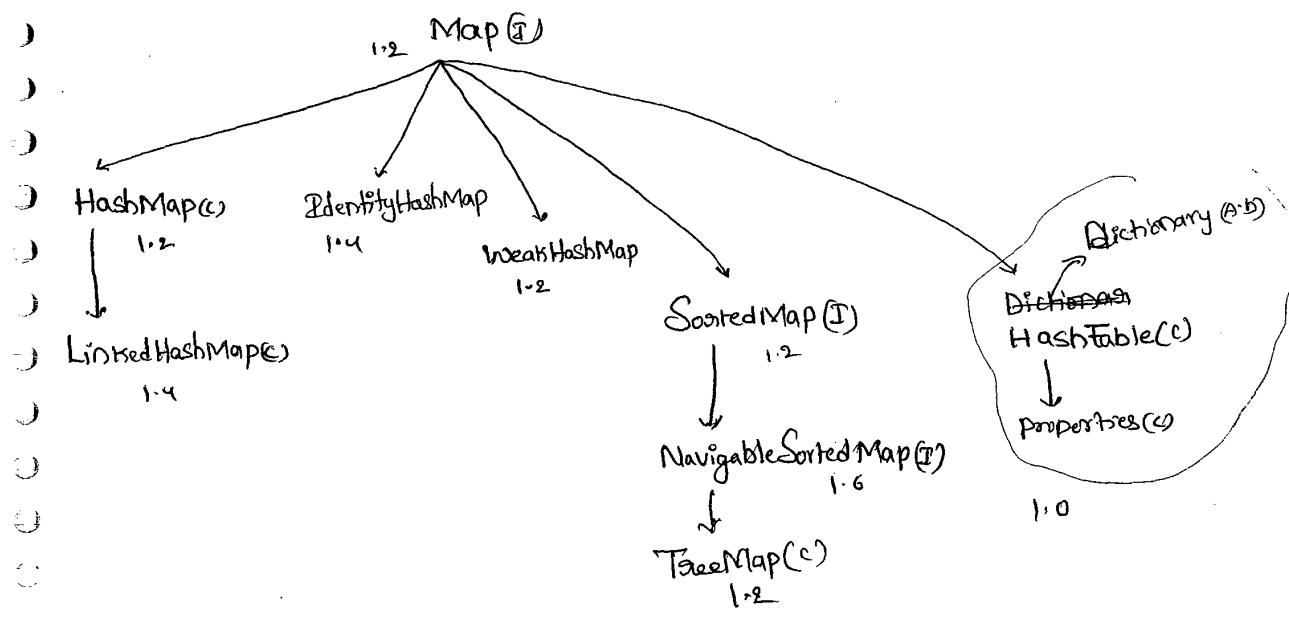
- If we want to represent a group of objects as key-value pairs then we should go for Map. Both Key & Value are Objects.
- Both Key & Values are Objects.
- Duplicate Keys are not allowed, But values can be duplicated.
- Each key-value pair is called Entry.

Ex:-

RollNo	Name
101	durga
102	Sainu
103	Ravi
104	Sambu
105	Sundar

Key → RollNo
value → Name
entry → Row

- There is no relationship b/w Collection & Map.
- Collection meant for a group of individual objects whereas
- Map meant for a group of key-value pairs.
- Map is not child interface of Collection.



Methods of Map Interface :-

* ① Object put(Object key, Object value);

→ To add Key-value pair to the map

→ If the Specified Key is already available then old value will be replaced with new value & old value will be returned.

② Void putAll(Map m)

→ To add a group of Key-value Pairs.

③ Object get(Object key)

→ Returns the value associated with Specified Key

→ If the Key is not available then we will get Null

④ Object remove(Object key);

⑤ boolean containsKey(Object key)

⑥ boolean containsValue(Object value)

⑦ int size();

⑧ boolean isEmpty();

⑨ Void clear();

⑩ Set keySet();

⑪ Collection values();

⑫ Set entrySet();

} Collection Views of the Map.

Entry (Interface)

- Each key-value pair is called One "Entry"
- Without existing Map Object There is no chance of Entry Object
- Hence, Interface Entry is define inside Map Interfaces.

Code:

interface Map

{

 interface Entry

{

 ①, Object getKey();

 ②, Object getValue();

 ③, Object setValue();

}

}

④ HashMap (c)

- The underlying data structure is HashTable
- Heterogeneous Objects are allowed for both Keys & values.
- Duplicate Keys are not allowed ~~for~~ but the values can be duplicated.
- Insertion Order is not preserved because it is based on HashCode of Keys.
- Null Key is allowed (only once)
- Null Values are allowed (any number of times).

* differences b/w HashMap & HashTable :-

HashMap	HashTable
① No method is Synchronized	① Every method is Synchronized
② multiple Threads Can Operate Simultaneously & Hence HashMap Object is not Thread Safe	② At a time Only one Thread is allowed to operate on ^{HashTable} object . Hence it is Thread Safe.
③ Threads are not required to wait & hence relatively performance is High.	③ It increases waiting time of the thread & hence performance is low.
④ null is allowed for both key & value	④ null is not allowed for Both key & values . otherwise we will get <u>NPEException</u>
⑤ Introduced in 1.2 version & it is Non-Legacy	⑤ Introduced in 1.0 version & it is Legacy

Q) How To get Synchronized Version of HashMap?

A) → By default HashMap object is not Synchronized, but we can get Synchronized Version by using SynchronizedMap() of Collections Class.

```
Map m = Collections.SynchronizedMap(HashMap hm);
```

Constructor :-

(i) `HashMap m = new HashMap();`

→ Creates an Empty `HashMap` object with default initial capacity level is 16 & default fillRatio 0.75 (75%).

(ii) `HashMap m = new HashMap(int initialCapacity)`

(iii) `-HashMap m = new HashMap(int initialCapacity, float fillRatio)`

(iv) `HashMap m = new HashMap(Map m)`

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

```

class HashMapDemo
{
    public static void main(String[] args)
    {
        HashMap m = new HashMap();
        m.put("chiranjeevi", 700);
        m.put("balaiyah", 800);
        m.put("venkatesh", 1000);
        m.put("nagarjuna", 500);
        System.out.println(m); } { venkatesh = 1000, balaiyah = 800, chiranjeevi = 700,
        nagarjuna = 500 }
        System.out.println(m.put("chiranjeevi", 1000)); 700
        Set s = m.keySet();
        System.out.println(s); [venkatesh, balaiyah, chiranjeevi, nagarjuna]
        Collection c = m.values();
        System.out.println(c); [1000, 800, 500, 700]
        Set S1 = m.entrySet();
        Iterator its = S1.iterator();
    }
}
```

```

while (its.hasNext())
{
}

```

Map.Entry m₁ = (map.Entry) its.next();

s.o.println(m₁.getKey() + " ---- " + m₁.getValue());

if (m₁.getKey().equals("nagajjuna"))

m₁.setValue(10000);

Nagarjuna	500
Venkatesh	1000
Balaiyah	800
Chiranjeevi	1000

s.o.println(m₁); } Nagajjuna = 10000, Venkatesh = 500, Balaiyah = 800,

Chiranjeevi = 1000

ii) LinkedHashMap :-

→ It is the child class of HashMap.

→ It is exactly same as HashMap except the following differences

HashMap	LinkedHashMap
① The underlying D.S is HashTable	① The underlying D.S is HashTable + Linked List
② Insertion Order is Not preserved	② Insertion Order is preserved
③ Introduced in 1.2 Version	③ Introduced in 1.4 Version

→ In the above program if we are replacing HashMap with LinkedHashMap, The following is the O/P.

{ chiranjeevi = 700, balaiyah = 800 } Venkatesh = 1000, Nagajjuna = 500 }

i.e insertion order is preserved

Notes -

→ The main application area of LinkedHashSet & LinkedHashMaps are cache applications implementation where duplication is not allowed & insertion order must be preserved.

(iii) IdentityHashMap :-

→ It is exactly same HashMap except the following difference.

→ In the case of HashMap to identify duplicate keys JVM always uses `equals()`, which is mostly meant for Content Comparison.

→ If we want to use `== operator` instead of `equals()` to identify duplicate keys we have to use IdentityHashMap. (`== operator` always meant for reference comparison).

Eg:- `HashMap m = new HashMap();`



`Integer i1 = new Integer(10);`



`Integer i2 = new Integer(10);`

`equals() → Content`
`== → reference`

`m.put(i1, "pavan");`

`I1 == I2 → False`

`m.put(i2, "Kalyan");`

`I1.equals(I2) → True`

`S.o.println(m); { 10 = Kalyan }`

→ In the above code `i1` & `i2` are duplicate keys because `i1.equals(i2)` returns true.

→ If we replace HashMap with IdentityHashMap then the o/p is

{ 10 = pavan , 10 = Kalyan }

→ `i1` & `i2` are not duplicate keys because `i1 == i2` returns false.

WeakHashMap :-

- It is exactly same as HashMap except the following difference.
- In the case of HashMap, Object is not eligible for g.c even though it doesn't have any external references if it is associated with HashMap. i.e., HashMap dominates Garbage Collector (g.c.).
- But in the case of WeakHashMap even though object associated with WeakHashMap, it is eligible for g.c. if it does not have any external references. i.e. G.c dominates WeakHashMap.

Eg:- import java.util.*;

class WeakHashMapDemo

{

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

 {

 HashMap m = new HashMap();

 Temp t = new Temp();

 m.put (t, "durga");

 S.o.println(m); // temp = durga}

 t = null;

 System.gc();

 Thread.sleep(5000);

 S.o.println(m);

}

 // temp = durga}

Class Temp

{

 public String toString()

{

 return "temp";

{

 public void finalize()

{

 System.out.println("finalize method called");

{

%! {temp = durga}

{temp = durga}

→ If we replace HashMap with WeakHashMap then the o/p is

{temp = durga}

finalize method Called

{}

(ii) Sorted Map (I) :-

- If we want to represent a group of entries according to some sorting order then we should go for SortedMap. The sorting should be done based on the keys but not based on the values.
- SortedMap interface is the child interface of Map.
- SortedMap interface defines the following 6 specific methods
 - ① Object firstKey();
 - ② Object lastKey();
 - ③ SortedMap headMap(Object key);
 - ④ SortedMap tailMap (Object key);
 - ⑤ SortedMap subMap (Object key1, Object key2);
 - ⑥ Comparator comparator();

(iii) TreeMap (II) :-

- The underlying D.S is RED-BLACK Tree,
- Insertion order is not preserved & all entries are inserted according to some Sorting Order of keys.
- If we are depending on default natural sorting order then the keys should be Homogeneous & Comparable. otherwise we will get ClassCastException (CE).
- If we are defining our own sorting order by Comparator then

The Keys Need not be Homogeneous & Comparable.

→ There are no restrictions on values, They can be Heterogeneous & Non-Comparable.

→ duplicate Keys are not allowed but values can be duplicated.

Null acceptance:-

→ For the Empty TreeMap as the first entry ^{with} null key is allowed but after inserting that entry if we are trying to insert any other entry we will get NullPointerException (NPE).

→ For the Non-Empty TreeMap if we are trying to insert entry with null key we will get NullPointerException (NPE)

→ There are no restrictions on null values. i.e., we can use null any no. of times anywhere for map values.

Constructors :-

(i) TreeMap t = new TreeMap()

for default natural sorting order.

(ii) TreeMap t = new TreeMap(Comparator c)

for customized sorting order.

(iii) TreeMap t = new TreeMap(Map m)

(iv) TreeMap t = new TreeMap(SelectableMap m)

Ex:- import java.util.*;

Class TreeMapDemo3

१

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

१

```
TreeMap m = new TreeMap();
```

```
m.put(100, "zzz");
```

m.put(103, "yyy");

m.put(101, "xxx");

```
m.put(104, 106);
```

```
m.put(107, null);
```

```
sm.putString("FFFF", "xxxx"); // CCE
```

// m.put(null, "xxx"); //NPE

$S \cdot o \cdot p10(m)$; } $100 = zzz$, $101 = xxx$, $103 = 444$, $104 = 106$, $107 = 0011$

1

O/P

$$\{100 = \text{zzz}, 101 = \text{xxx}, 103 = \text{yyy}, 104 = \text{106}, 107 = \text{null}\}$$

Eg:-

```
import java.util.*;
```

```
class TreeMapDemo
```

```
{
```

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

```
{
```

```
        TreeMap t = new TreeMap(new MyComparator());
```

```
        t.put("xxx", 10);
```

```
        t.put("AAA", 20);
```

```
        t.put("zzz", 30);
```

```
        t.put("LLL", 40);
```

```
        System.out.println(t);
```

```
}
```

```
class MyComparator implements Comparator
```

```
{
```

```
    public int compare(Object obj1, Object obj2)
```

```
{
```

```
        String s1 = obj1.toString();
```

```
        String s2 = obj2.toString();
```

```
        return s2.compareTo(s1);
```

```
}
```

Q/P:-

$\left. \begin{array}{l} zzz = 30, \ xxx = 10, LLL = 40, AAA = 20 \end{array} \right\}$

Hashtable():

- The underlying datastructure is HashTable.
- Heterogeneous objects are allowed for both keys & values
- Insertion order is not preserved & it is based on hashCode of the keys.
- null is not allowed for both key & values otherwise we will get NullPointerException (NPE).
- duplicate keys are not allowed, but values can be duplicated.
- All methods are Synchronized & hence HashTable object is ThreadSafe.

Constructor:

(i) Hashtable h = new Hashtable()

→ Creates an Empty Hashtable Object with default initial capacity is 11 & default fillratio 75% (0.75).

(ii) Hashtable h = new Hashtable(int initialCapacity)

(iii) Hashtable h = new Hashtable(int ^{initialCapacity}, float ^{fillratio})

(iv) Hashtable h = new Hashtable(Map m);

Eg:- import java.util.*;

Class HashtableDemo

{

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

}

Hashtable h = new Hashtable();

h.put(new Temp(5), "A");

h.put(new Temp(2), "B");

h.put(new Temp(6), "C");

h.put(new Temp(15), "D");

h.put(new Temp(23), "E");

h.put(new Temp(16), "F");

// h.put("duarga", null); //NPE

System.out.println(h);

}

} { 6=C, 16=F, 5=A, 15=D, 2=B, 23=E }

Class Temp

{

int i;

Temp(int i)

{

this.i = i;

}

Public int hashCode()

{

Return i;

}

Public String toString()

{

Return i + " ";

}

10	
9	
8	
7	
6	6=C
5	5=A, 16=F
4	15=D
3	
2	2=B
1	23=E
0	

From top to bottom & Right to Left

Properties :-

- It is the child class of Hashtable
- In our program if anything which changes frequently (like database usernames, passwords, URL) never recommended to hardCode the value in the Java program. Because for every change, we have to recompile, rebuild, redeploy the application & sometime even server restart also required. which creates a big business impact to the client.
- we have to configure those variables (properties) inside properties files & we have to read those values from java code.
- the main advantage of this approach is, If any change in the properties file just redeployment is enough which is not a business impact to the client.

Constructor :-

(i) Properties p = new Properties();

→ In the case of Properties both key & value should be String type

Methods :-

* (i) String getProperty(StringPropertyName)

→ Returns the value associated with specified property.

(ii) String setProperty(String pName, String pValue);

→ to set a new property.

(iii) String Enumeration getPropertyNames();

* (iv) void load(InputStream is)

→ To load the properties from properties files into java properties-object.

(v) void store(OutputStream os, String comment)

→ To update properties from properties object into properties file.

```

Eg:- import java.util.*;
import java.io.*;
class PropertiesDemo
{
    public static void main(String[] args) throws IOException
    {
        Properties p = new Properties();
        FileInputStream fis = new FileInputStream("abc.properties");
        p.load(fis);
        System.out.println(p);
        String s = p.getProperty("Venki");
        System.out.println(s);
        p.setProperty("Nag", "999999");
        FileOutputStream fos = new FileOutputStream("abc.properties");
        p.store(fos, "updated by dunga for SCJP Demo class");
    }
}

```

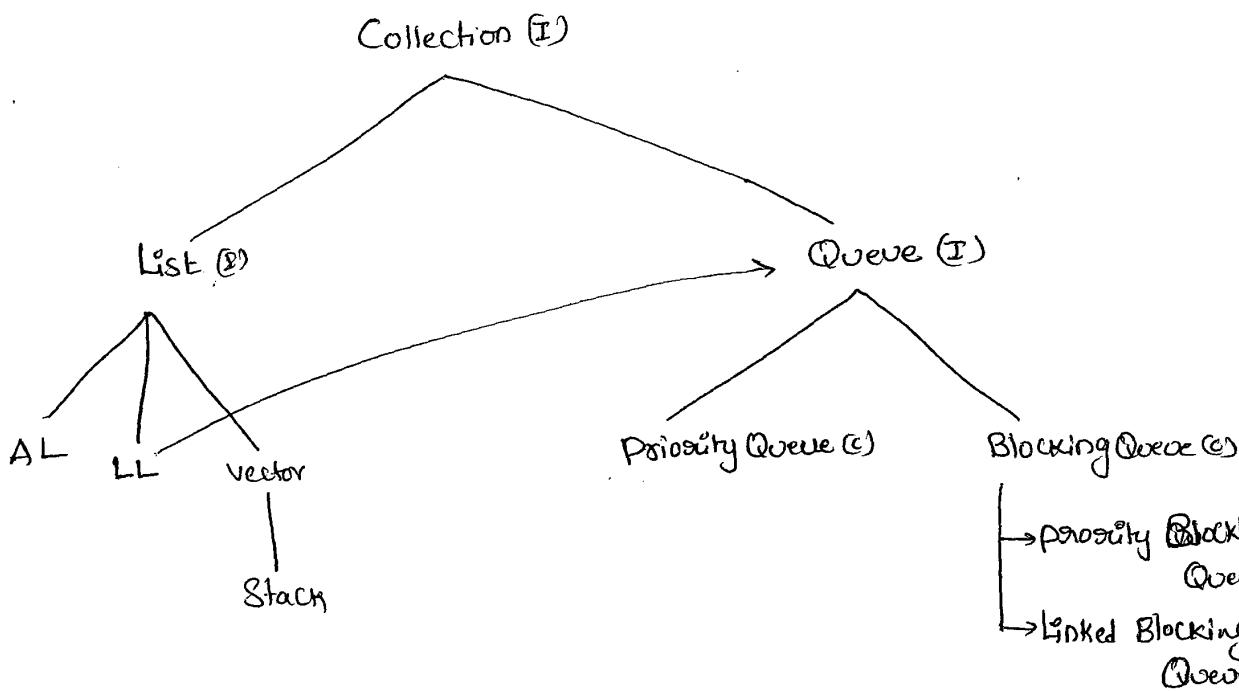
User = Scott
Venki = 8888
Prod = tiger

abc.properties

1.5 Version Enhancement :-

Queue (I) :-

- It is the child Interface of Collection.
- If we want to represent a group of individual objects power to processing. Then we should go for Queue.



- Usually Queue follows FIFO (first in first out), But Based on our requirement we can change our order.
- From 1.5 Version onwards LinkedList implements Queue Interface.
- LinkedList Based implementation of Queue always follows FIFO

Queue Interface methods :-

(i) boolean offer(Object obj)

→ To add an object into the Queue.

(ii) Object peek();

→ To return head element of the Queue. If Queue is Empty then

This method returns null.

(iii) Object element();

→ To return head element of the Queue. If Queue is Empty

Then we will get RuntimeException Saying NoSuchElementException

(iv) Object poll();

→ To remove & return head element of the Queue. If Queue

is Empty then this method returns null.

(v) Object remove();

→ To remove & return head element of the Queue, if Queue is

Empty then we will get RuntimeException Saying NoSuchElementException

Priority Queue :-

- This is the DataStructure to hold a group of individual Objects prior to processing According to Some priority.
- The priority can be either default natural Sorting order or Customized Sorting order.
- If we are depending on default Natural Sorting Compulsory objects should be Homogeneous & Comparable otherwise we will get ClassCastException.
- If we are defining our own Customized Sorting by Comparator then the objects need not be Homogeneous & Comparable.
- Duplicate objects are not allowed.
- Insertion order is not preserved.
- Null insertion is not possible even as first element also.

Constructors :-

(i) Priority Queue q = new Priority Queue();

→ Creates an Empty Priority Queue with default initialCapacity 11
& Priority Order is default natural Sorting order.

(ii) Priority Queue q = new Priority Queue(int initialCapacity);

(iii) Priority Queue q = new Priority Queue(int initialCapacity, Comparator c);

(iv) Priority Queue q = new Priority Queue(Collection c);

(v) Priority Queue $q = \text{new Priority Queue}(\text{SortedSet } s)$

Eg:-

```

import java.util.*;
class PriorityQueueDemo
{
    public static void main(String[] args)
    {
        PriorityQueue q = new PriorityQueue();
        System.out.println(q.peek()); //null
        //System.out.println(q.element()); //NSE noSuchElementException
        for(int i=0 ; i<=10 ; i++)
        {
            q.offer(i);
        }
        System.out.println(q); // [0, 1, 2, 3, 4, 5, ----- 10]
        System.out.println(q.poll()); // 0
        System.out.println(q); // [1, 2, 3, 4, 5 ----- 10]
    }
}
  
```

Eg 2:-

```

import java.util.*;
class PriorityQueueDemo2
{
    public static void main(String[] args)
    {
    }
}
  
```

PriorityQueue q = new PriorityQueue(15, new MyComparator());

q.offer("A");

q.offer("Z");

q.offer("L");

q.offer("B");

System.out.println(q); // [Z, L, B, A]

{ }

Class MyComparator implements Comparator

{

public int compare(Object obj1, Object obj2)

{

String s1 = (String) obj1;

String s2 = obj2.toString();

return s2.compareTo(s1);

{ }

Output: [Z, L, B, A]

1.6 Version Enhancements :-

(i) NavigableSet (I) :-

- It is the child interface of SortedSet.
- This interface defines several methods to provide support for navigation for the TreeSet object.
- The following list of various methods present in NavigableSet.

(i) ceiling(e) :-

→ Returns the lowest element which is $\geq e$.

(ii) higher(e) :-

→ Returns the lowest element which is $> e$.

(iii) floor(e) :-

→ Returns highest element which is $\leq e$.

(iv) lower(e) :-

→ Returns the highest element which is $< e$.

(v) pollFirst() :-

→ Remove & returns first element

(vi) pollLast() :-

→ Remove & returns last element.

(vii) descendingSet() :-

→ Returns the NavigableSet in reverse order.

```
Eg: import java.util.*;  
class NavigableSetDemo  
{  
    public static void main(String[] args)  
{  
        TreeSet<Integer> t = new TreeSet<Integer>();  
        t.add(1000);  
        t.add(2000);  
        t.add(3000);  
        t.add(4000);  
        t.add(5000);  
        System.out.println(t); // [1000, 2000, 3000, 4000, 5000]  
        System.out.println(t.ceiling(2000)); // 2000  
        System.out.println(t.higher(2000)); // 3000  
        System.out.println(t.floor(3000)); // 3000  
        System.out.println(t.lower(3000)); // 2000  
        System.out.println(t.pollFirst()); // 1000  
        System.out.println(t.pollLast()); // 5000  
        System.out.println(t.descendingSet()); // [5000, 4000, 3000, 2000]  
        System.out.println(t); // [2000, 3000, 4000]  
    }  
}
```

(ii) NavigableMap (I):-

→ It is the child interface of SortedMap to define several methods for navigation purposes.

→ The following is the list of methods present in NavigableMap.

(i) ceilingKey(e)

(ii) higherKey(e)

(iii) floorKey(e)

(iv) lowerKey(e)

(v) pollFirstEntry()

(vi) pollLastEntry()

(vii) descendingMap()

Eg:-

```
import java.util.*;
```

```
class NavigableMapDemo
```

```
{
```

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

```
{
```

```
TrieMap<String, String> t = new TrieMap<String, String>();
```

```
t.put ("b", "banana");
```

```
t.put ("c", "cat");
```

```
t.put ("a", "apple");
```

```
t.put ("d", "dog");
```

```
t.put ("g", gun);
```

```
S. o. p. n (t); { a=apple, b=banana, c=cat, d=dog, g=gun }
```

S.o.println(t.ceilingKey("c")); c
S.o.println(t.higherKey("e")); g
S.o.println(t.floorKey("e")); d
S.o.println(t.lowerKey("e")); d
S.o.println(t.pollFirstEntry()); a = apple
S.o.println(t.pollLastEntry()); g = gun
S.o.println(t.descendingMap()); } d = dog , c = cat , b = banana }
S.o.println(t); } b = banana , c = cat , d = dog }

Collections class

Collections class:-

- It is an utility class present in java.util package
- It defines several utility methods for collection implemented class objects

Sorting the elements of a list :-

- Collections class defines the following methods to sort elements of a List.

① Public static void Sort(List l) :-

- We can use these method to sort according to Natural Sorting Order.
- In this case Comparing elements should be Homogeneous & Comparable. otherwise we will get ClassCastException.
- List should not contain null, otherwise we will get NullPointerException

② Public static void Sort(List l, Comparator c) :-

- To sort elements of a List according to Customized Sorting order

Searching the elements of a list :-

- Collections class defines the following method to search elements of a List

① Public static int binarySearch(List l, Object obj)

- If the List is sorted according to natural sorting order then we have to use this method.

⑨ public static int binarySearch(List l, Object key, Comparator c)

→ If the List is Sorted according to Comparator Then we have to use this method.

Conclusion :-

- Internally binarySearch method uses Binary Search algorithm.
- Before Calling binarySearch() method Compulsory the List should be Sorted otherwise we will get unpredictable results.
- Successfull Search returns index.
- Unsuccessfull Search returns insertion point
- Insertion point is the Location where we can place element in the Sorted List.
- If the List is Sorted according to Comparator Then at the time of Search also we should pass the Same Comparator Otherwise we will get Unpredictable results.

Ex:- To Search elements of list

import java.util.*;

class CollectionsSearchDemo

↓

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

↓

ArrayList l = new ArrayList();

l.add("z");

l.add("A");

l.add("m");

`l.add("K");`

`l.add("a");`

`S.o.println(l); [z, A, M, K, a]`

`Collections.sort(l);`

`S.o.println(l); [A | K | M | z | a]`

-1	-2	-3	-4	-5	-6
o	1	2	3	4	
A	K	M	Z	a	

`S.o.println(Collections.binarySearch(l, "z")); 3`

`S.o.println(Collections.binarySearch(l, "j")); -2`

}

Ex:-

`import java.util.*;`

`class CollectionsSearchDemo1`

`{`

`P.S.V.m()`

`ArrayList l = new ArrayList();`

`l.add(15);`

`l.add(0);`

`l.add(20);`

`l.add(10);`

`l.add(5);`

`S.o.println(l); [15 | 0 | 20 | 10 | 5]`

`Collections.sort(l, new MyComparator());`

`S.o.println(l); [20 | 15 | 10 | 5 | 0]`

`S.o.println(Collections.binarySearch(l, 10, new MyComparator())); //2`

`S.o.println(Collections.binarySearch(l, 13, new MyComparator())); // -3`

`S.o.println(Collections.binarySearch(l, 17)); // -6 unpredictable`

}

because it is not passing `Comparator()`

-1	-2	-3	-4	-5	-6
0	1	2	3	4	
20	15	10	5	0	

```

Class MyComparator implements Comparator {
    public int compare(Object obj1, Object obj2)
    {
        Integer i1 = (Integer) obj1;
        Integer i2 = (Integer) obj2;
        return i2.compareTo(i1);
    }
}

```

Note :-

→ For the List Contains n elements Range of Successfull Search

① Range of Successfull Search : 0 to $n-1$

② Range of unsuccessful Search : $-(n+1)$ to -1

③ total Range : $-(n+1)$ to $n-1$

Ex:-

-1	-2	-3	-4
10	20	30	
0	1	2	

Range of successful Search : 0 to 2

Range of unsuccessful Search : -4 to -1

Total Range : -4 to 2

Reversing the elements of a List :-

→ Collections class defines The following reverse method for this

Public static void reverse(List l);

Ex:- To Reverse elements of List

```
import java.util.*;
```

```
Class CollectionsReverseDemo
```

```
{
```

```
    P. S. v. m(       )
```

```
}
```

```
    AL l = new AL();
```

```
    l.add(15);
```

```
    l.add(0);
```

```
    l.add(20);
```

```
    l.add(10);
```

```
    l.add(5);
```

```
    S.o.println(l);     15 | 0 | 20 | 10 | 5
```

```
    Collections.reverse(l);
```

```
    S.o.println(l);     5 | 10 | 20 | 0 | 15
```

```
}
```

reverse() Vs reverseOrder():-

- We can use reverse() method to reverse the elements of a list and this method contains List arguments.
- Collections class defines reverse order method also to return Comparator object for reversing original sorting order.

Comparator c = Collections.reverseOrder(Comparator c)



- reverseOrder() method contains Comparator argument whereas reverse() contains List arguments.

Ex:- To REVERSE ELEMENTS OF LIST

```
import java.util.*;  
class CollectionsReverseDemo  
{  
    public static void main(String[] args)  
    {  
        ArrayList l = new ArrayList();  
        l.add(15);  
        l.add(0);  
        l.add(20);  
        l.add(10);  
        l.add(5);  
        System.out.println(l); // 15 | 0 | 20 | 10 | 5  
        Collections.reverse(l);  
        System.out.println(l); // 5 | 10 | 20 | 0 | 15  
    }  
}
```

Arrays Class

Arrays Class :-

→ It is an utility class present in Util package, To define Several utility methods for Arrays for both primitive Arrays & Object type Arrays.

Sorting the elements of Array :-

→ Arrays class defines the following methods for this.

① public static void Sort (primitive[] p);

→ To Sort elements of ^{primitive} Array According to Natural Sorting order.

② public static void Sort (Object[] a)

→ To Sort elements of Object Array According to Natural Sorting Order.

→ In this Case Compulsory the elements should be Homogeneous & Comparable. Otherwise we will get ClassCastException.

③ public static void Sort (Object[] a, Comparator c)

→ To Sort elements of Object[] according to Customized Sorting order.

Note :-

primitive Arrays can be Sorted only by natural Sorting order

whereas Object arrays can be Sorted either by natural Sorting Order or by Customized Sorting Order.

Ex:- To SORT elements of Arrays

ArraysSortDemo.java

```
import java.util.Arrays;  
import java.util.Comparator;
```

```
Class ArraysSortDemo
```

↓

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

↓

```
int[] a = {10, 5, 20, 11, 6};
```

```
S.o.println("primitive Array before Sorting:");
```

```
for(int ai : a)
```

↓

```
S.o.println(ai);
```

↓

10
5
20
11
6

```
Arrays.sort(a);
```

```
S.o.println("primitive Array After Sorting:");
```

```
for(int ai : a)
```

↓

```
S.o.println(ai);
```

↓

5
10
11
20

```
String[] s = {"A", "Z", "B"};
```

```
S.o.println("Object Array Before Sorting:");
```

```
for(String a2 : s)
```

↓

```
S.o.println(a2);
```

↓

A
Z
B

```
Arrays.sort(s);
```

```
S.o.println("Object Array After Sorting:");
```

↓

for (String ai : s)

 ↓
 S.o.println(ai); A
 B

}
Arrays.sort(s, new MyComparator());

S.o.println("Object Array After Sorting by Comparator:");

for (String ai : s)

 ↓
 S.o.println(ai); B
 A

}
Class MyComparator implements Comparator {

 public int compare(Object o1, Object o2) {

 String s1 = o1.toString();

 String s2 = o2.toString();

 return s2.compareTo(s1);

}
}

Searching The Elements of Array:-

→ Arrays class defines the following search methods for this.

① public static int binarySearch(primitive() p, primitive key)

② public static int binarySearch(Object() o, Object key)

③ public static int binarySearch(Object() o, Object key, Comparator c)

Notes:-

All rules of these binarySearch() method are Exactly same as Collections class binarySearch() method.

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

`import static java.util.Arrays.*;`

`class ArraysSearchDemo`

`}`

`P.S.V.M()`

`{`

`int[] a = {10, 5, 20, 11, 6};`

`Arrays.sort(a); // Sort by natural order`

-1	-2	-3	-4	-5	-6
5	6	10	11	20	
0	1	2	3	4	

`S.o.println(Arrays.binarySearch(a, 6)); // 1`

`S.o.println(Arrays.binarySearch(a, 14)); // -5`

`String[] s = {"A", "z", "B"},`

`Arrays.sort(s);`

-1	-2	-3	-4
A	Z	B	
0	1	2	

`System.out.println(Arrays.binarySearch(s, "z")); // 2`

`S.o.println(Arrays.binarySearch(s, "S")); // -3`

`Arrays.sort(s, new MyComparator());`

-1	-2	-3	-4
2	B	A	
0	1	2	

`S.o.println(Arrays.binarySearch(s, "z", new MyComparator())); // 0`

`S.o.println(Arrays.binarySearch(s, "S", new MyComparator())); // -2`

`S.o.println(Arrays.binarySearch(s, "N")); // unpredictable result`

`}`

`class MyComparator implements Comparator`

`{`

`public int compare(Object o1, Object o2)`

`{`

```

String S1 = O1.toString();
String S2 = O2.toString();
return S2.compareTo(S1);
}
}

```

Converting Arrays to List :-

① Public static List asList(Object[] a)

→ By Using this method we are not Creating an independent List Object

just we are Creating List view for the existing Array Object.

→ By using List reference if we perform any operation the changes

will be reflected to the Array reference. Similarly, By using Array

reference if we perform any changes those changes will be reflect

to the List.

→ By Using List reference we can't perform any operation which varies

the size, (i.e., add & remove) otherwise we will get RuntimeException

Saying "Unsupported - Operation - Exception" (UOE).

→ By using List reference we can perform replacement operation

But replacement should be with the same-type of element only otherwise

We will get RuntimeException Saying "ArrayStoreException"

Ex:- To view Array IN LIST FORM.

ArrayListDemo.java

```
import java.util.*;
```

```
class ArraysAsListDemo
```

```
{
```

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

```
{
```

```
        String[] s = {'A', 'z', 'B'};
```

```
        List l = Arrays.asList(s);
```

```
        System.out.println(l); // [A, z, B]
```

```
        s[0] = 'K';           [A, z, B]  [K, z, B]
```

```
        System.out.println(l); // [K, z, B]
```

```
        l.set(1, "L");      [K, L, B]
```

```
        for (String s1 : s)
```

```
            System.out.println(s1); // [K, L, B]
```

```
        l.add("dogar");      R.E // USE
```

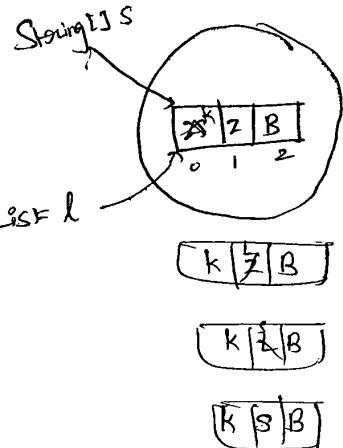
```
        l.remove(2);         R.E // USE
```

```
        l.set(1, "S");       [K, S, B]  => [K, S, B]
```

```
        l.set(1, 10);        R.E // ArrayStoreException
```

```
}
```

```
}
```



185

卷之三

- 1) Introduction
- 2) Generic Classes
- 3) Bounded types
- 4) Generic methods
- 5) Wild Card Characters
- 6) Communication with non-Generic Code.
- 7) Conclusions.

Introduction:

- Arrays are always Safe w.r.t type.
- for Example, if our programme requirement is -to add only String Objects then we can go for String[] array. For this array we can add only String type of objects, by mistake if we are trying to add any other type we will get Compiletime Error.

Ex:- String[] s = new String[600];

s[0] = "durga"; ✓

Type-Safe.

s[1] = "Paran"; ✓

s[2] = new Student(); X



C.E:- incompatible types

↳ found Student
required: String

→ Hence In The Case of Arrays we can always give the guarantee about the Type of elements. String[] array Contains only String Objects. (reString) due to this arrays are always Safe to use w.r.t type.

→ But Collections are not Safe to use w.r.t type. For Example if our programme requirement is to hold only String Objects & if we are using ArrayList, By mistake if we are trying to add any other type to the List we won't get any Compiletime-Error But program may fail at Runtime.

Ex:-

```
ArrayList l = new ArrayList();
```

```
l.add("durga");
```

```
l.add("sainu");
```

```
l.add(new Student());
```

```
!
```

✓ String name1 = (String)l.get(0);

✓ String name2 = (String)l.get(1);

✗ String name3 = (String)l.get(2);



R.E!: ClassCastException.

→ There is no guarantee that Collection can Hold a particular type of objects. Hence w.r.t type Collections are not Safe to use.

Cases :-

- In the Case of Arrays at the time of retrieval it is not required to perform any TypeCasting.

Ex:- `String[] s = new String[600];
s[0] = "durga";
 |`

`String name1 = s[0];`

TypeCasting is not required.

- But in the Case of Collections at the time of retrieval Compulsory we should perform TypeCasting otherwise we will get CompiletimeError.

Ex:- `ArrayList l = new ArrayList();`

`l.add("durga");
 |`

`String name1 = l.get(0);`

c.e!. Incompatible type
found : object

Required : String

But

`String name1 = (String)l.get(0);` ✓

- Hence, in the Case of Collections TypeCasting is mandatory which is a bigger headache to the programmer.

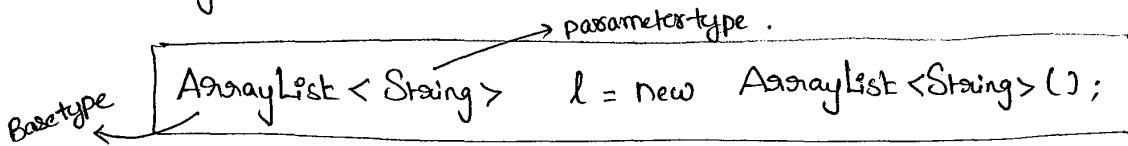
- To Overcome the above problems of Collections (TypeSafe & TypeCasting) Sun people introduced Generics Concepts in 1.5 Version. Hence The main objectives of Generic Concepts are,

Hence the main objectives of Generic Concepts are,

1) To provide Type Safety to the Collections. So that they can hold Only a particular Type of Objects.

2) To resolve Type Casting problems.

→ For Example → to Hold only String Type of Objects a Generic version of ArrayList we can declare as follows.


Base type ArrayList < String > parameter type

→ For this ArrayList we can add only String type of Objects, by mistake if we are trying to add any other type we will get Compiletime Error. i.e., we are getting Type-Safety.

L.add("durga"); ✓
L.add("Sainu"); ✓
L.add("10"); ✓
L.add(10); ✗ C.E: - Cannot find Symbol
Symbol : method add (int)
location : class ArrayList<String>

→ At the time of retrieval it is not required to perform any Type Casting.

String name = L.get(0); ✓

↓
Type Casting is not required

Conclusion 1 :-

- Usage of parent class reference to hold child class objects is considered as polymorphism.
- Polymorphism concept is applicable only for base type, but not for parameter type.

Ex:- ↗ Base type ↗ Parameter type.

```

    ✓ ArrayList < Integer > l = new ArrayList < Integer > ();
    ✓ List < Integer > l = new ArrayList < Integer > ();
    ✓ Collection < Integer > l = new ArrayList < Integer > ();
    ✗ List < Object > l = new ArrayList < Integer > ();
  
```

CE! - Incompatible types

→ Found : ArrayList<Integer>
Required : List<Object>

Conclusion 2 :-

- For the parameter-type we can use any class or interface name.
- If we can't use primitive type, violation leads to Compiletime Error.

Ex:- ArrayList < int > l = new ArrayList < int > ();

CE!
Unexpected type
→ Found : int
Required : Reference

CE!
Unexpected type
→ Found : int
Required : Reference

Generic - classes :-

→ Until 1.4v a non-Generic Version of ArrayList Class is declared as follows.

Class ArrayList

↓

Add (Object o);

Object get(int index)

↓

→ The assignment to the add() method is Object. Hence we can add any type of object due to this we are not getting Type-Safety.

→ The return type of get() method is Object, hence at the time of retrieving Compulsory we should perform Type Casting.

→ But in 1.5v a Generic Version of ArrayList Class is declared as follows.

class ArrayList<T>
 ^
 Type parameter.

↓
add <T t>

T get (int index)

↓

→ Based on our runtime requirement Type parameter 'T' will be replaced with Corresponding provided type.

→ for example, To hold only String type of object we have to Create Generic Version of ArrayList Object as follows.

`ArrayList<String> l = new ArrayList<String>();`

→ for this requirement the corresponding loaded version of ArrayList Class is,

```
Class ArrayList<String>
{
    add (String s)
    String get (int index)
}
```

- add() method Can take String as the assignment hence we can
- add only String type of objects. By mistake if we are trying to add any other type we will get `CompiletimeError`. i.e., we are getting Type-Safety.
- The return type of get() method is String, Hence at the time of retrieval we can assign directly to the String type variable it is not required to perform any TypeCasting.

Note :-

- 1. As the Type parameter we can use any valid java identifier but it is Convention to use "T". e

Ex:-	Class AL < x >	Class AL < Dog >
✓	↓	✓

Q) we can pass any no. of type parameters but & need not be one class

Ex:- Class HashMap<K, V>

}

{

HashMap<String, Integer> m = new HashMap<String, Integer>();
key type
value type

→ Through Generics we are associating a type parameter to the classes. Such type of parameterized classes are called Generic classes.

→ we can define our own Generic classes also.

Ex:-

Class Gen<T>

{

T ob;

Gen(T ob)

{

this.ob = ob;

{

public void show()

{

S.O.P("The type of ob is :" + ob.getClass().getName());

{

public T getOb()

{

return ob;

{

Class GenDemo

}

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

}

Gen<String> g₁ = new Gen<String> ("durga");

①

g₁.show(); // the type of ob is : java.lang.String
System.out.println(g₁.getOb()); durga

Gen<Integer> g₂ = new Gen<Integer> (10);

②

g₂.show(); // the type of ob is : java.lang.Integer.
System.out.println(g₂.getOb()); 10

}

Bounded Types:

→ we can bound the type parameters for a particular range by

using extends keyword.

Ex:-

Class Test<T>

↓

↳

→ As the type parameter we can pass any type hence it is unbounded type.

✓ Test<String> t₁ = new Test<String>();

✓ Test<Integer> t₂ = new Test<Integer>();

Ex 2: Class Test<T extends Number>
 |
 y

→ As the type parameter we can pass either Number type or its child classes. It is bounded type.

✓ Test<Integer> t₁ = new Test<Integer>();

✗ Test<String> t₂ = new Test<String>();

C.E!: Type parameter java.lang.String is not with in its bound

→ We Can't Bound Type Parameter By using implements & Super Keywords :

Ex 1:- ① Class Test<T implements Runnable>
 X
 |
 y

X ② Class Test<T Super Integer>
 |
 y

But,

→ implements keyword purpose we can survive by letting Extends keyword only

Ex 1. Class Test<T extends X>
 |
 y
 |
 class / interface.

→ x → Can be either class / interface.

→ if x is a class then as the type parameter we can provide either x type or its child classes.

→ if x is an interface as the type parameter we can provide either x type or its implementation classes.

Ex:- Class Test<T extends Runnable>

↓

↳

✓ Test<Runnable> t₁ = new Test<Runnable>();

✓ Test<Thread> t₂ = new Test<Thread>();

✗ Test<String> t = new Test<String>();

↖
C.E!-

Type parameter java.lang.String is not within its Bound

→ We can bound the type parameter even in combination also.

Ex:-

Class Test<T extends Number & Runnable>

→ As the type parameters we can pass any type which is the child class of Number & implements Runnable interface.

① class Test<T extends Runnable & Comparable>

✓ ② class Test<T extends Number & Runnable & Comparable>

✗ ③ class Test<T extends Number & Thread>

→ We can't extend more than

one class at a time.

➤ ⑤ Class Test < T extends Runnable & Number >

→ we have to take first class & Then interface.

Generic Methods & Wild Card Characters?

→ ① m₁ (ArrayList<String> l) ✓

→ This method is applicable for ArrayList<String> (ArrayList of only String type).

→ Within the method we can add String-type objects & null to the list if we are trying to add any other type we will get Compilation Error.

Ex:- m₁ (ArrayList<String> l)
↓
l.add ("A"); ✓
l.add (null); ✓
l.add (10); ✗

② m₁ (ArrayList < ? extends X > l) ✓

→ we can call this method by passing ArrayList of any-type. But within the method we can't add any-type except null to the list. Because we don't know the type exactly.

③ Ex:- m₁ (ArrayList< ? > l)
↓
l.add (null); ✓
l.add ("A"); ✗
l.add (10); ✗

3) $m_1(Al < ? \text{ extends } X > l)$ ✓

→ If X is a class then we can call this method by passing

ArrayList of either X type or its child classes.

→ If X is an interface then we can call this method by passing

ArrayList of either X type or its implementation class

→ In this case also we can't add any type of elements to the list

Except null

4) $m_1(ArrayList < ? \text{ Super } X > l)$ ✓

→ If X is a class then this method is applicable for ArrayList

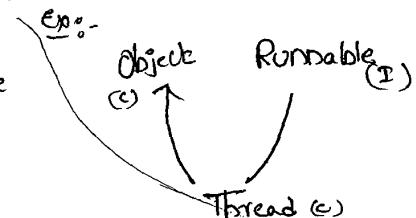
of either X type or its Super classes.

→ If X is an interface then this method is applicable for ArrayList

of either X type or Super classes of implementation class of X

→ Within the method we can add only X type

Objects & null to the List



Q) Which of the following declarations are valid?

✓ ① $AL < String > l = new AL < String >();$

✓ ② $AL < ? > l = new AL < String >();$

✓ ③ $AL < ?, \text{ extends } String > l = new AL < String >();$

✓ ④ $AL < ? \text{ Super } String > l = new AL < String >();$

✓ ⑤ $AL < ? \text{ extends } Object > l = new AL < String >();$

✓ ⑥ $\text{AL} < ? \text{ extends Number} >$ $l = \text{new AL} < \text{Integer} > ()$;

✗ ⑦ $\text{AL} < ? \text{ extends Number} >$ $l = \text{new AL} < \text{String} > ()$;

C.E!: Incompatible types

found: $\text{AL} < \text{String} >$

required: $\text{AL} < ? \text{ extends}$

$\text{Number} >$

✗ ⑧ $\text{AL} < ? >$ $l = \text{new AL} < ? \text{ extends Number} > ()$;

✗ ⑨ $\text{AL} < ? >$ $l = \text{new AL} < ? > ()$,

C.E!: unexpected type

found: ?

required: Class or interface without
bounds.

→ We can define the type parameter either at class-level or
at method-level.

Declaring type parameter at class level!

class Test <T>

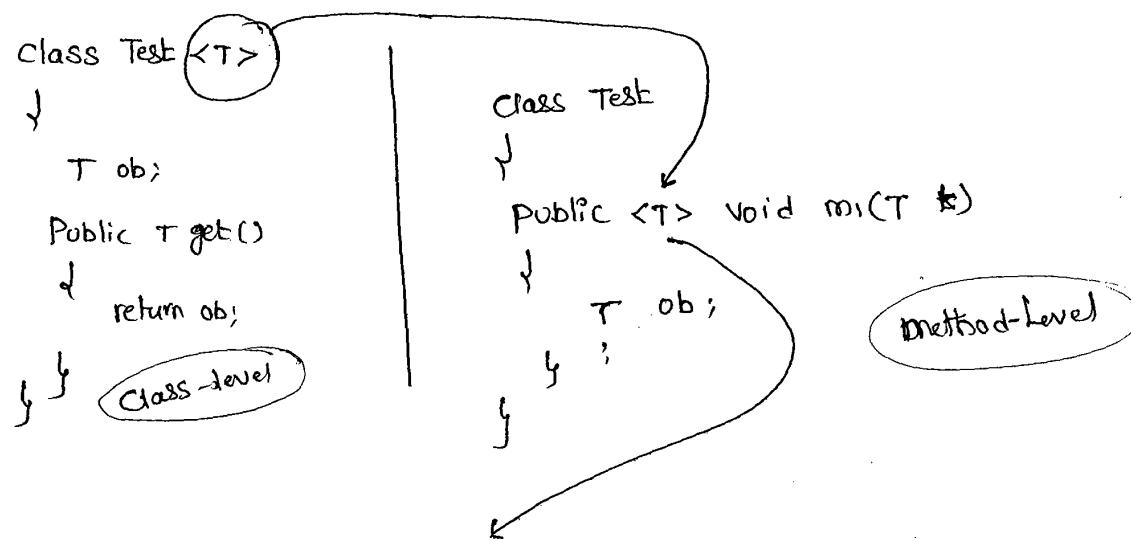
T ob;

public T get()

return ob;

Declaring Type parameter at method-level:-

→ we have to declare the type parameter just before the return type.



- ✓ ① <T extends Number>
- ✓ ② <T extends Runnable>
- ✓ ③ <T extends Number & Runnable>
- ✓ ④ <T extends Runnable & Comparable>
- ✗ ⑤ <T extends Number & Thread>
- ✗ ⑥ <T extends Runnable & Thread>

Communication with non-Generic Code :-

- To provide compatibility with old version SUN people compromised.
- The concept of Generics in very few areas. The following is one such area.

Ex:- Class Test

```

    ↓
    p. S. v. m( String[] args)
    ↓
    AL<String> l = new AL<String>();
    l.add("A");
}

```

Ex:- Class Test

Generic area

```

    ↓
P. S.v.m(—)
    }

AL<String> l = new AL<String>();
l.add("A");
l.add(10); C.E
m(l);
S.o.println(l); [A, 10, 10.5, true]
// l.add(10); C.E
  
```

Non-Genetic
area

```

    ↓
static
public void m(AL l)
    }

l.add(10); ✓
l.add(10.5 10.5); ✓
l.add(true);
  
```

Conclusions :-

- ④ Generics Concepts are applicable only at Compiletime to provide type Safety & to resolve type Casting problems. At Runtime there is no Suchtype of Concept. Hence the following declarations are equal,

Ex:-

all are equal	✓	AL l = new AL();
	✓	AL l = new AL<String>();
	✓	AL l = new AL<Integer>();

Ex:- ArrayList l = new ArrayList<String>();

19/11/52

l.add("A"); ✓

l.add(10); ✓

l.add(true); ✓

System.out.println(l); [A, 10, true]

→ The following two declarations are equal & there is no difference

both are
equal

1) AL<String> l = new AL<String>();

2) AL<String> l = new AL();

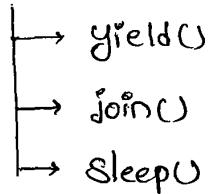
196

卷之三

09/04/2011

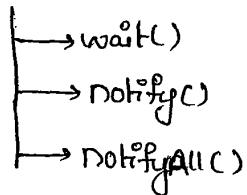
Multithreading

- ① Introduction
- ② The ways to define, instantiate, and start a thread
- ③ Getting & Setting name of a thread
- * ④ Thread properties
- ⑤ The methods to prevent thread execution



- * ⑥ Synchronization

- ⑦ Interthread Communication



- ⑧ Deadlock

- ⑨ Daemon threads

Multitasking :-

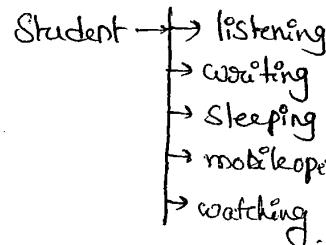
→ Executing Several tasks Simultaneously is called "Multitasking".

There are 2 types of multitasking.

(1) process - based multitasking.

(2) thread - based multitasking.

Ex:- Students in Class Room.



(1) process-based multitasking:-

→ Executing Several tasks Simultaneously, where each task is a Separate independent process, is called process based multitasking.

Ex:- While typing a Java program in editor we can able to listen audio songs by mp3 player in the System. at the same time we can download a file from the net. all these tasks are executing simultaneously & independent of each other. Hence, it is process-based multitasking.

→ process-based multitasking is best Suitable at "O.S Level".

(2) thread-based multitasking:-

→ Executing Several tasks Simultaneously where each task is a Separate independent part of The Same program is called "thread based multitasking" & each independent part is called "thread".

→ It is Best Suitable for "programmatic level".

→ whether it is process-based or thread-based the main objective of multitasking is to improve performance of the system by reducing response time.

→ the main important application areas of multithreading are developing video games, multimedia graphics, implementing animations....

→ Java provides inbuilt support for multithreading by introducing a rich API (Thread, Runnable, ThreadGroup, ThreadLocal....). Being a programmer we have to know how to use this API and we are not responsible to define that API. Hence, developing multithreading programs is very easy when compared with C++.

② The ways to define, instantiate & start a new thread :-

→ we can define a thread in the following 2 ways.

(i) By extending Thread class.

(ii) By implementing Runnable interface.

Defining a thread by extending Thread class :-

defining a thread by Extending Thread class :-

Ex:-

```

class Mythread extends Thread
{
    public void run()
    {
        for(int i=0; i<=10; i++)
        {
            System.out.println("child thread");
        }
    }
}

```

defining a thread

Job of thread

executing child thread
(mythread)

Class ThreadDemo

```

class ThreadDemo
{
    public static void main(String[] args)
    {
        Mythread t = new Mythread(); // instantiation of thread
        t.start(); // starting of a thread
    }
}

for(int i=0; i<=10; i++)
{
    System.out.println("main thread");
}

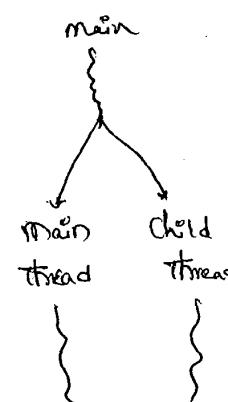
```

main thread →

child thread →

Two threads →

executing main thread ←



Case 1:-Thread Scheduler :-

→ When ever multiple threads are waiting to get chance for execution which thread will get chance first is decided by Thread Scheduler whose behaviour is JVM vendor dependent. Hence we can't expect exact execution orders & hence exact O/p.

→ Thread Scheduler is the part of JVM. Due to this unpredictable behaviour of Thread Scheduler we can't expect exact O/p for the above program. The following are various possible O/p.

<u>P-1</u>	<u>P-2</u>	<u>P-3</u>	<u>P-4</u>
main thread	child thread	child thread	main thread
		main thread	main
		main thread	
child thread	main thread		child
		child thread	
			main thread
		main thread	

Note:-

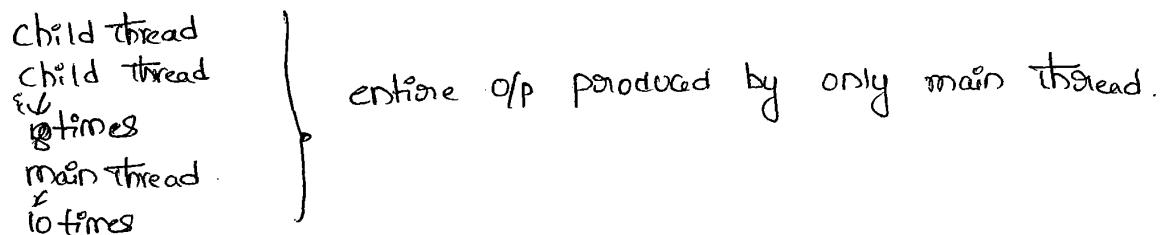
→ When ever the situation comes to multithreading the guarantee in behaviour is very less. We can tell possible O/p but not exact O/p.

Case 2:-Difference b/w t.start() & t.run():

→ In the case of t.start() a new thread will be created & that thread is responsible to execute run().

- But in the case of `t.start()` no new thread will be created & `run` method will be executed just like a normal method call.
- In the above program, if we are replacing `t.start()` with `t.run()` the following is the O/P.

O/P:-



Case 3:-

Importance of Thread class `start()` method!

- To start a thread, the required mandatory activities (like - registering thread with Thread Scheduler) will be performed automatically by Thread class `start()` method. Because this facility, programmer is not responsible to perform this activity & he is just responsible to define job of the thread. Hence Thread class `start()` plays very important role & without executing that method there is no chance of starting a new thread.

Q:-

class Thread

 |

 start()

- * 1. Register this thread with Thread Scheduler & perform other initialization activities

 |

`t.run()`

 |

Case 4:-

→ If we are Not overriding run() method :-

→ If we are not overriding run() method, then Thread class run() will be executed which has Empty implementation & Hence we won't get any o/p.

Ex:-

```
class Mythread extends Thread
{
}
class ThreadDemo
{
    public static void main(String[] args)
    {
        mythread t = new Mythread();
        t.start();
    }
}
O/P:- no o/p pointing
```

Note:-

* It is highly recommended to override run() to define our Job.

Case 5:-

Overloading of run() :-

→ Overloading of the run() is possible, but Thread class start() will always call no argument run() only. but the other run() we have to call explicitly just like a normal method call.

Ex:- class mythread extends Thread

```
    {
        public void run()
        {
            System.out.println("run()");
        }
        public void run(int i)
        {
            System.out.println("run(int i)");
        }
    }
```

Class ThreadDemo

```
    {
        public static void main(String[] args)
        {
            mythread t = new mythread();
            t.start();
        }
    }
```

O/P:- run()

Case 6:-

Overriding of start() :-

→ If we override start() then start() will be executed just like a normal method call & no new thread will be created.

Ex:- class mythread extends Thread

```
    {
        public void start()
    }
```

S. o. pIn ("Start method")

}

public void run()

{

S. o. pIn ("run");

}

class ThreadDemo

{

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

{

MyThread t = new MyThread();

t.start();

}

O/P:- Start method.

Case(F):-

class MyThread extends Thread

{

public void start()

{

Super.start();

S. o. pIn ("Start method");

}

public void run()

{

S. o. pIn ("run");

}

;

Class ThreadDemo

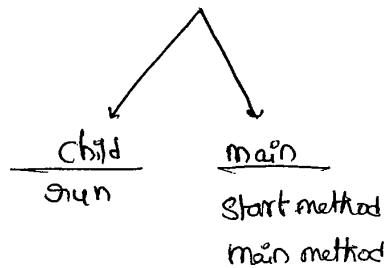
↓
p.s.v.m(String[] args)

↓
MyThread t = new MyThread();

t.start();

S.o.p("main method");

}



O/P:-

P-1 ✓

P-2 ✓

P-3 ✓

P-4 X

Start method

run

Start method

main method

run

Start method

Start method

Main method

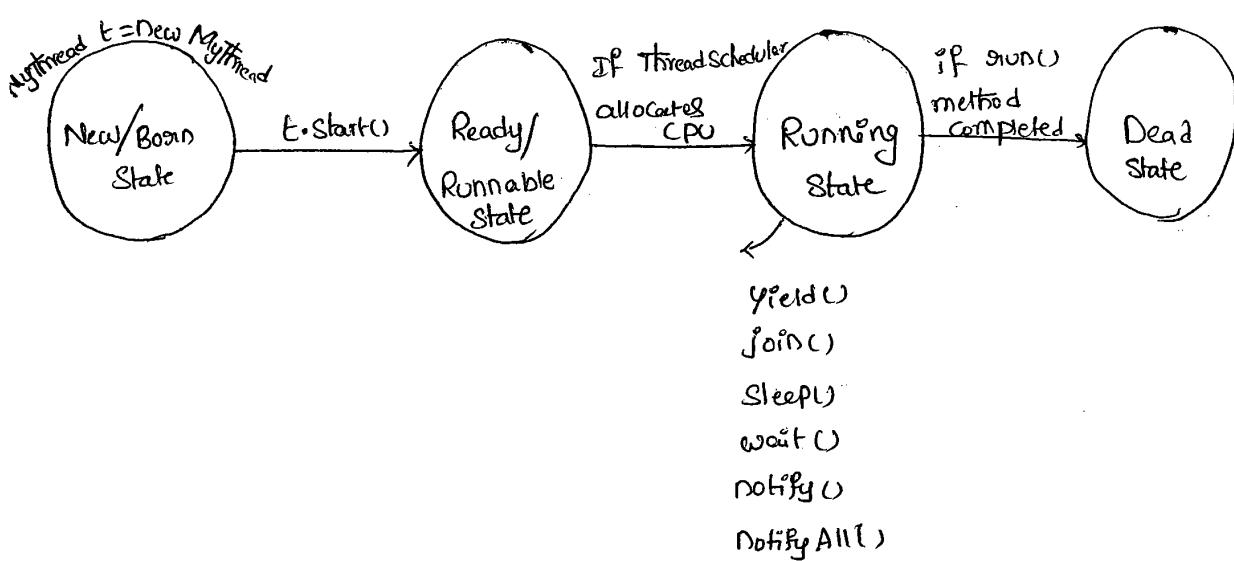
Main method

run

run

Case-8:-

* Life Cycle of a Thread :-



→ Once we Created a Thread Object Then it is Said to be in

New State or born State.

→ If we Call `start()` method then the Thread will be ^{entered} into Ready or Runnable State.

→ If ThreadScheduler allocates CPU, then the Thread will entered into Running State.

→ If `run()` method Completes then the Thread will entered into DeadState

* Case 9:-

→ After Starting a Thread we are not allowed to Restart the Same Thread once again otherwise we will get Illegal Runtime Exception saying "IllegalThreadStateException".

e.g.:

Thread t = new Thread()

t.start();

;

t.start(); X R.E! - IllegalThreadStateException. (ITSE)

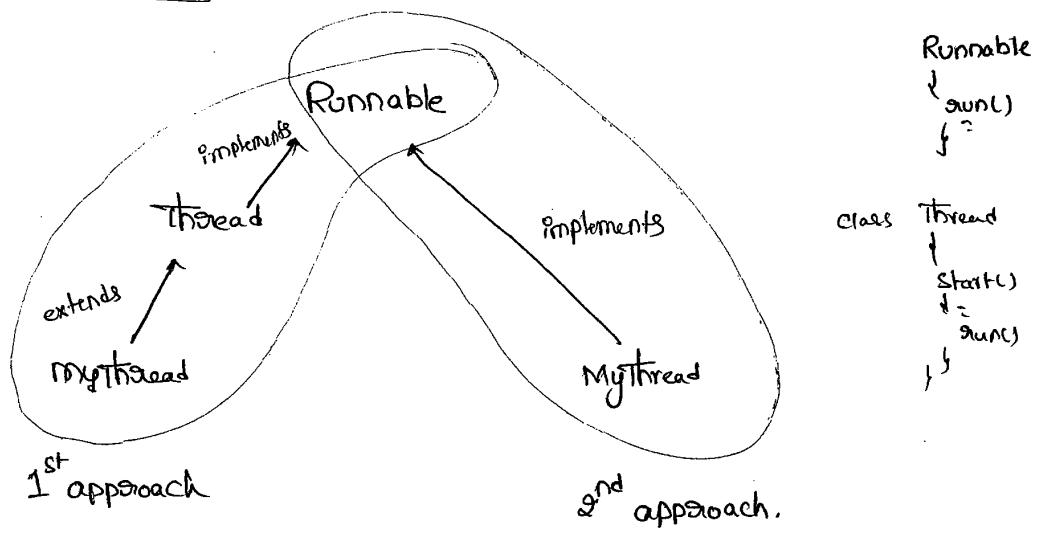
→ Within the `run()` if we Call `super.start()` we will get the Same Run-time Exception.

Note:-

→ It's Never Recommended to override `start()`, but it is highly Recommended to override `run()`.

(2) defining a thread by implementing "Runnable Interface":-

- * We can define a thread even by implementing Runnable Interface also.
- * Runnable Interface present in Java.lang package & Containing ONLY one method run() method.



Ex:-

Class MyRunnable implements Runnable

```
public void run()  
{  
    for(int i=0 ; i<10 ; i++)  
    {  
        System.out.println("child Thread");  
    }  
}
```

defining a thread class

Job of Thread

Class Thread Demo

↓

p.s.v.m (Strong I args)

↓

MyRunnable $\sigma_1 = \text{new MyRunnable();}$

Thread $t = \text{new Thread}(\sigma_1);$

\hookrightarrow target Runnable

$t.start();$

→
for (int i=0; $i < 10$; $i++$)

↓

$\text{System.out.println("main thread");}$

↓

{ } { }

→ We Can't get Exact o/p & we will get mixing o/p

Case Study :

MyRunnable $\sigma_1 = \text{new MyRunnable();}$

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

Thread $t_2 = \text{new Thread}(\sigma_1);$

Case(1) :-

(i) $t_1.start();$

→ A new Thread will be Created which is responsible for Execution
of Thread class run().

Case(2) :- $t_2.run();$

→ No New Thread will be Created & Thread class run() will be Executed Just Like a Normal method call.

Case 3:- $t_2.start()$:

→ New Thread will be Created which is Responsible for the Execution of MyRunnable run() method.

Case 4:- $t_2.run()$:

→ No new Thread will be Created & MyRunnable run() will be Executed just like a normal method Call.

Case 5:- $g_1.start()$:

→ We will get Compiletime Error Saying Start() is not available in MyRunnable class

C:E!, Cannot find Symbol

Symbol : method start()

location : class myRunnable

Case 6: $g_1.run()$:

→ No new Thread will be Created & MyRunnable run() will be Executed Just like a normal method Call.

Q) In which of the above cases a new Thread will be created

A) $t_1.start() \in t_2.start()$

Q) In which of the above Case MyRunnable class run() will be Executed Just like a normal method?

$t_2.run() \neq g_1.run()$

Best Approach to define a Thread :-

- Among the two ways of defining a thread implements Runnable mechanism is recommended to use.
- In the first approach, Thread our class always extending Thread class & hence there is no chance of extending any other class. But in the second approach we can extend some other class also while implementing Runnable interface. Hence 2nd approach is recommended to use.

Thread Class Constructors :-

- ① Thread t = new Thread();
- ② Thread t = new Thread(Runnable g);
- ③ Thread t = new Thread(String name);
- ④ Thread t = new Thread(Runnable g, String name);
- ⑤ Thread t = new Thread(ThreadGroup g, String name);
- ⑥ Thread t = new Thread(ThreadGroup g, Runnable g);
- ⑦ Thread t = new Thread(ThreadGroup g, Runnable g, String name);
- ⑧ Thread t = new Thread(ThreadGroup g, Runnable g, String name, long stackSize);

Durga's approach to define a Thread (not recommended to use).

Ex:- Class Mythread extends Thread

{

 public void run()

{

 System.out.println("run method");

}

Class Test

{

 public static void main(String[] args)

{

 Mythread t = new Mythread();

 Thread t1 = new Thread(t);

 t1.start();

 } // System.out.println("main");

}

O/P:-

run |
main
main | run

3) Getting & Setting name of a thread :-

→ Every Thread in Java has Some Name. It may be provided by the programmer or default name generated by JVM.

→ We Can get & Set name of a Thread by using the following methods of Thread class.

- (i) public final String getName();
- (ii) public final void setName(String name);

Ex:-

```

class Test
{
    public static void main(String[] args)
    {
        System.out.println(Thread.currentThread().getName()); // main
        Thread.currentThread().setName("Parabag");
        System.out.println(Thread.currentThread().getName()); // parabag
    }
}

```

Note!:-

→ we Can get Current Executing Thread Reference by using the following method of Thread class.

```
public static Thread currentThread();
```

4) Thread priority:-

- Every thread in Java has Some priority but the range of Thread priority is "1 to 10". (1 is least & 10 is highest).
- Thread class defines the following Constants to define Some Standard priorities.

- 1) Thread.MIN_PRIORITY → 1
- 2) Thread.NORM_PRIORITY → 5
- 3) Thread.MAX_PRIORITY → 10
- 4) Thread.LOW_PRIORITY X
- 5) Thread.HIGH_PRIORITY X

- Thread Scheduler will use these priorities while allocating Cpu
- The Thread which is having highest priority will get chance first.
- If Two threads having Same priority then we Can't expect Exact Execution order, it depends on Thread Scheduler.

default priority:-

- The default priority only for the main thread is 5.
But for all the Remaining threads it will be Inheriting from the parent. i.e whatever the priority parent has the same priority will be inheriting to the child.

- * Thread class defines the following 2 methods to get & set priority of a thread,

① public final int getPriority();
 ② public final void setPriority(int p);

→ the allowed values are 1 to 10, otherwise we will get IllegalArgumentException.

Ex:-

t.setPriority(5); ←

t.setPriority(10); ←

✗ t.setPriority(100); ✗ R.E :- IAE (IllegalArgumentException).

Ex:-

Class Mythread extends Thread

↓

public void run()
 {

for(int i=0; i<10; i++)

 System.out.println("Child Thread");

}

Class ThreadPriorityDemo

{
 public static void main(String[] args)

 Mythread t = new Mythread();

 // t.setPriority(10); → ①

 t.start();

 for(int i=0; i<10; i++)

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

→ If we are Commenting line① Then Both main & child threads having the Same priority (5) & Hence we can't expect Exact Execution order and Exact o/p.

→ If we aren't Commenting line① then main thread has the priority 5 & child thread has the priority 10 & Hence child thread will be Executed first & then main thread. In this case the o/p is

child thread	≡	6 times
main thread	≡	10 times

* The methods to prevent Thread Execution :-

→ We can prevent a thread from execution by using the following methods.

- (i) yield()
- (ii) join()
- (iii) sleep()

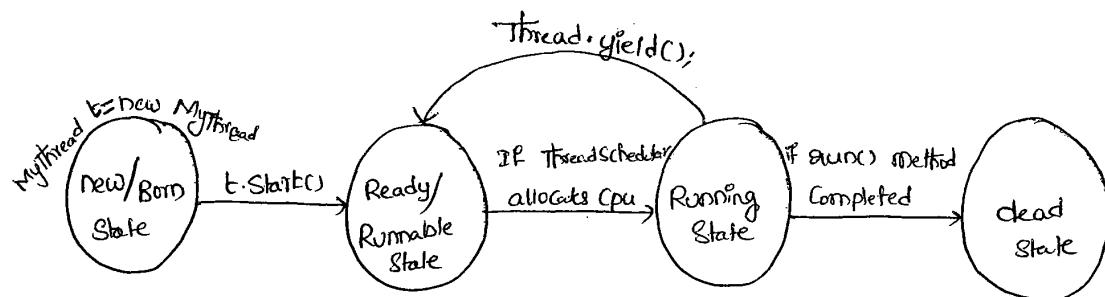
(i) yield() :-

→ yield() method causes, to pause Current executing thread for giving the chance to remaining waiting threads of Same priority.

→ If there are no waiting threads or all waiting threads have low priority then the same thread will continue it's execution once again.

→ Signature of yield method

```
public static void native void yield()
```



→ The Thread which is yielded, when it will get chance once again for execution is decided by threadscheduler. & we can't expect exactly.

Ex:- Class Mythread extends Thread

```

public void run()
{
    for (int i=0 ; i<10 ; i++)
        Thread.yield();           ----- ①
    System.out.println("child thread");
}
  
```

```
class ThreadYieldDemo
```

```

public static void main(String[] args)
{
    Mythread t = new Mythread();
    t.start();
    for (int i=0 ; i<10 ; i++)
        System.out.println("main thread");
}
  
```

→ If we are Commenting Line① The both threads will be Executed

Simultaneously & we Can't Expect Exact Execution Order.

→ If we are Not Commenting Line① Then the chance of Completing main Thread first is high because child Thread always calls yield().

ii) join() :-

→ If a Thread wants to wait until Completing Some Other Thread Then we should go for join() method.

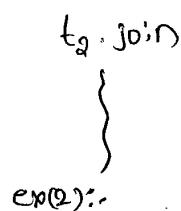
Ex:- i) view fixing (t₁)



Cards pointing (t₂)

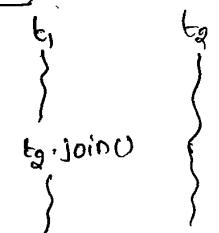


Cards distributing (t₃)



exp:-

→ If Thread T₁ Executes t₂.join() Then t₁ thread will entered into waiting state until t₂ Completes. Once t₂ Completes then t₁ will Continue its execution.

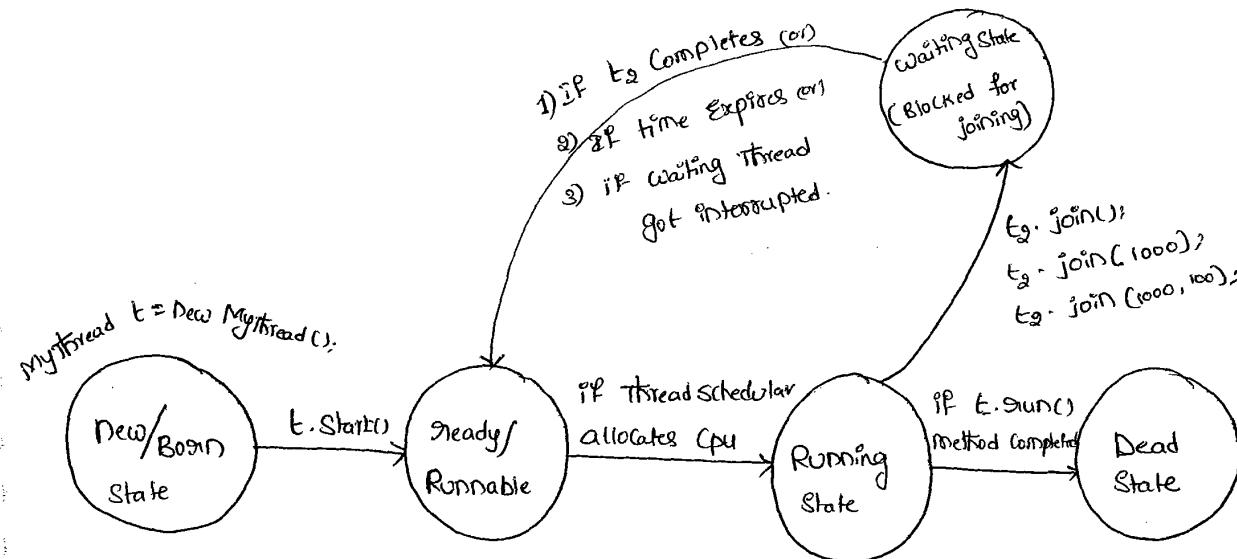


(i) public final void join() throws InterruptedException

(ii) public final void join(long ms) throws InterruptedException

(iii) public final void join(long ms, int ns) throws InterruptedException,

→ join() method is Overloaded and Only join() throws InterruptedException. Hence, when ever we are using join() Complusay we should handle InterruptedException, either by try-catch or by throws other wise we will get Compiletime Error.



Class MyThread extends Thread

```

}
public void run()
{
    for(int i=0 ; i<10 ; i++)
        System.out.println("Sitha Thread");
}
```

```

    try
    {
        Thread.sleep(2000);
    }
    catch(IE e)
    {
    }
}
```

Class ThreadJoinDemo

```

}
public class ThreadJoinDemo
{
    public static void main(String[] args) throws InterruptedException
```

```

    MyThread t1 = new MyThread();
    t1.start();
    t1.join(); → ①
}
```

```

for(int i=0 ; i<10 ; i++)
{
    S.o.println("Rama Thread");
}

```

→ If we are Commenting Line① Then both threads will be Executed Simultaneously and we Can't Expect Exact Execution Order. And Hence we can't Expect Exact o/p.

→ If we are not Commenting Line① then main Thread will wait until Completing child thread. Hence in this case the o/p is Expected.

O/P :-

SitaThread 10 times	≡
RamThread 10 times	≡
≡	≡

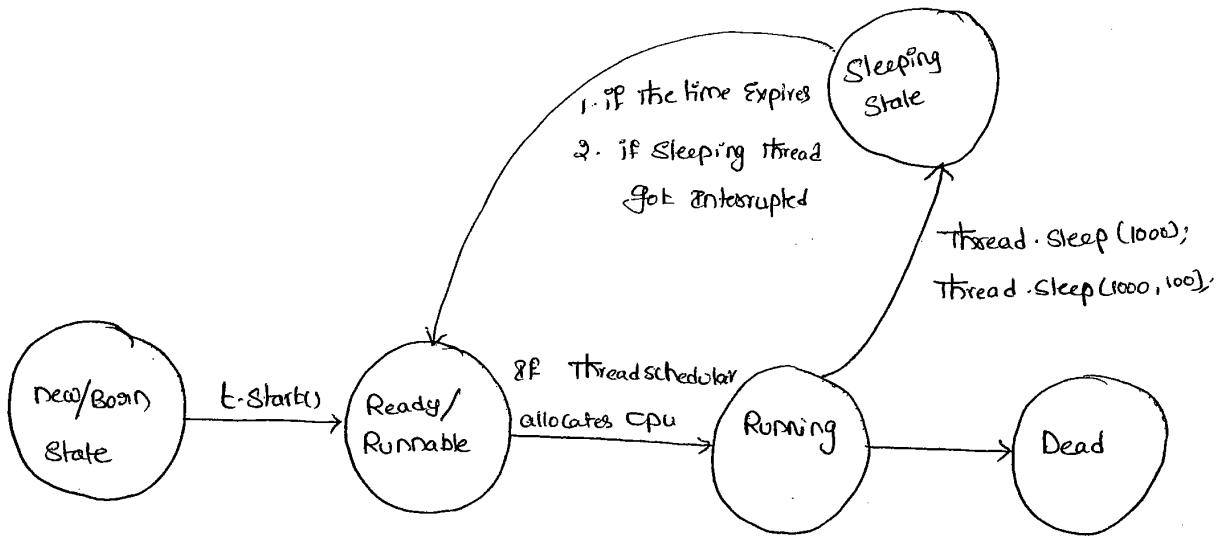
(iii) Sleep() :-

→ If a Thread don't want to perform any operation for a particular amount of time (Just pausing) Then we should go for Sleep().

- Public Static void Sleep(long ms) throws InterruptedException
- Public Static void Sleep(long ms, int ns) throws InterruptedException

→ When ever we are using Sleep method Compulsory we should Handle InterruptedException otherwise we will get Compile-time Error.

Static! because sleep method calls Thread.Sleep() means class name b.start(); b, is object so it is instance (a) non-static



Ex:- class Test

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

S. o. p(" Durga");

Thread. Sleep(5000);

S. o. p(" Software");

Thread. Sleep(5000);

S. o. p(" Solutions");

}

Interruption of a Thread :-

- * A Thread Can Interrupt another Sleeping or Waiting Thread.
- * for this Thread class defines interrupt() method.

```
public void interrupt()
```

Ex:- Class MyThread extends Thread

```
    public void run()
    {
        try
        {
            for (int i=0 ; i<100 ; i++)
                System.out.println("Lazy Thread");
            Thread.sleep(5000);
        }
        catch (IE e)
        {
            System.out.println("I got Interrupted");
        }
    }
}

Class InterruptDemo
{
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.start();
        → t.interrupt(); → ①
        System.out.println("end of main");
    }
}
```

→ If we are Commenting line① Then main Thread Won't Interrupt Child Thread Hence both threads will be executed until Completion

→ If we are not Commenting line① Then main Thread Interrupts the Child Thread hence ~~child thread won't Cont~~ causes Interrupted Exception.

→ In This Case the o/p is

O/P: 8 am Lazy Thread

I got Interrupted

End of main

Note:-

^{mayn't}

→ We ~~can't~~ See the Impact of interrupt Call immediately.

→ When ever we are Calling interrupt() method, if the target Thread is not in Sleeping or waiting State then there is no impact immediately.
Interrupt Call will wait until target Thread entered into Sleeping or waiting State. Once target Thread entered into Sleeping or waiting State the interrupt Call will impact the target Thread.

* Comparison Table for yield(), join(), Sleep() :-

Property	yield()	join()	Sleep()
i) Purpose ?	to pause current executing thread to give the chance for the remaining threads of same priority.	if a thread want to wait until completing some other thread then we should go for join	If a thread don't want to perform any operation for a particular amount of time (pausing) go for sleep()
ii) Static	Yes	No	Yes
iii) Is it overloaded	No	Yes	Yes
iv) Is it final	No	Yes	No
v) Is it throws InterruptedException	No	Yes	Yes
vi) Is it native method	Yes	No	Sleep (long ms) ↳ native Sleep (long ms, int ms) ↳ non-native

Synchronization :-

→ Synchronized is the modifier applicable only for methods & blocks
& we can't apply for classes & variables.

→ If a method or block declared as Synchronized then at a time
only one thread is allowed to execute that method or block on the
given object.

→ The main advantage of Synchronized key-word is we can resolve
data inconsistency problem.

→ The main limitation of Synchronized keyword is it increases
waiting time of the threads & effects performance of the system.
Hence if there is no specific requirement it's never recommended to
use Synchronized key-word.

→ Every object in Java has a unique lock synchronization concept
internally implemented by using this Lock concept. whenever we are
using synchronization then only Lock concept will come into the picture.

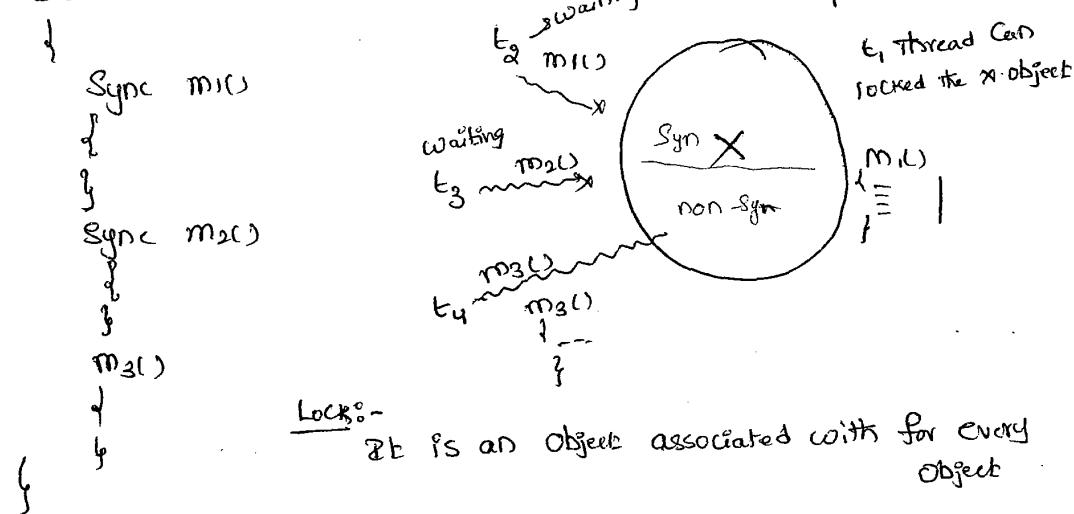
→ If a thread wants to execute any Synchronized method on the given
object, first it has to get the lock of that object. Once a thread
get a lock then it allowed to execute any Synchronized method
on that object.

→ Once Synchronized method completes then automatically the lock will be
released.

→ While a thread executing any synchronized method on the given object the remaining threads are not allowed to execute any synchronized method on the given object ~~simultaneously~~.

But remaining threads are ^{allowed to} execute any non-synchronized methods simultaneously (lock concept is implemented based on object but not based on method).

Ex:- Class X



Ex:-

```

Class Display
{
    public void wish(String name)
    {
        for (int i = 0; i < 10; i++)
        {
            System.out.print("Good morning: ");
            try
            {
                Thread.sleep(3000);
            }
            catch (Exception e)
        }
    }
}

```

```

S.o.println(name);
}
}

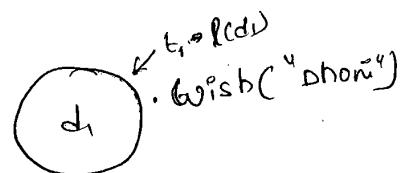
class MyThread extends Thread
{
    Display d;
    String name;

    MyThread(Display d, String name)
    {
        this.d = d;
        this.name = name;
    }

    public void run()
    {
        d.wish(name);
    }
}

class SynchronizedDemo
{
    public static void main(String[] args)
    {
        Display d1 = new Display();
        MyThread t1 = new MyThread(d1, "Dhoni");
        MyThread t2 = new MyThread(d1, "Yuvraj");
        t1.start();
        t2.start();
    }
}

```



→ If we are not declaring `wish()` method as Synchronized then both Threads will be Executed Simultaneously & we Can't Expect Exact O/p we will get irregular O/p.

O/p:-
Goodmorning; Goodmorning : Dhoni
Goodmorning : yuvraj
" : Dhoni
" "

→ If we declare `wish()` method as Synchronized then threads will be Executed One by one So that we will get regular O/p

O/p:-
Goodmorning : Dhoni
! 10times
Goodmorning : yuvraj
! 10times

Case Study :-

`Display d1 = new Display();`

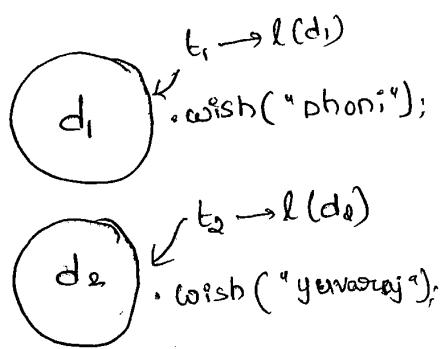
`Display d2 = new Display();`

`MyThread t1 = new MyThread(d1, "Dhoni");`

`MyThread t2 = new MyThread(d2, "yuvraj");`

`t1.start();`

`t2.start();`



→ Even though `wish()` method is Synchronized we will get irregular O/P in this case. Because, the threads are operating ~~on~~ different Objects.

Reason:-

→ Whenever multiple threads are operating on same Object then only Synchronization play the role. If multiple threads are operating on multiple Objects then there is no impact of Synchronization.

Classlevel Locks:-

→ Every class in Java has a unique lock,

→ If a thread wants to Execute a Static Synchronized method then it required classlevel lock.

→ While a Thread executing a Static Synchronized method then the remaining threads are not allowed to execute any Static Synchronized method of that class simultaneously but remaining threads are allowed to execute the following methods simultaneously.

- ✓ 1. Normal Static Methods.
- ✓ 2. Normal instance methods.
- ✓ 3. Synchronized instance methods.

Ex:- Class X

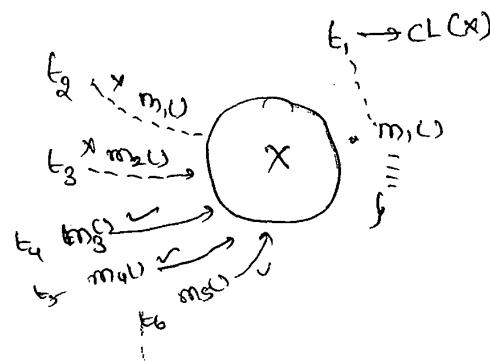
Static Syn `m1()`

Static Syn `m2()`

Syn `m3()`

Static `m4()`

`m5()`



Note:-

- There is no link between Object Level lock & Class Level Lock both are independent of each other.
- ClassLevel lock is different & ObjectLevel lock is different.

Synchronized Block :-

- if very few lines of code requires Synchronization then it is never recommended to declare entire method as synchronized we have to declare those few lines of code inside synchronized block.
- the main Advantage of Synchronized Block over Synchronized method is, it reduces the waiting time of the threads & improves performance of the system.

Ex(1) :-

- we can declare synchronized block to get current object lock as follows.

Synchronized (this)

{

:

}

- if thread got lock of current object then only it is allowed to execute this block.

Ex(2) :-

- To get lock of a particular object b we can declare synchronized block as follows,

20/30

Synchronized(b)

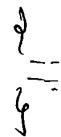


→ If Thread got lock of 'b' Then only it is allowed to Execute That blocks.

Ex(3):

→ To Get Class level lock we can declare Synchronized block as follows.

Synchronized(classname.class)



→ If Thread got Classlevel lock of classname (ex Display) class Then only it is allowed to Execute that block.

Ex(4):

Synchronized block Concept is applicable only for Objects & classes but not for primitives otherwise we will get Compiletime Error.

```
int x=10;  
Synchronized(x)  
{  
    ...  
}
```

C.E:- Unexpected type

found: int

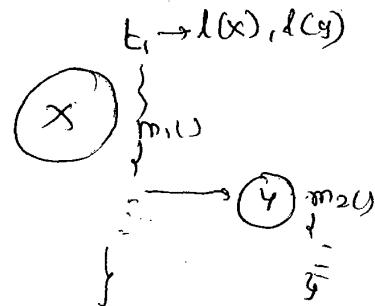
Required: Reference

→ Every object in java has a unique lock, But a thread can acquire more than one lock at a time (ofcourse from diff. objects)

Ex:- Class X

```
{  
    Syn m1();  
}  
y y = new Y();  
y.m2();  
y}
```

Class Y
↓
Syn m2();
{
 }
y



FAQ:-

- ① Explain about Synchronized Keyword & what are various Advantages & disadvantages?
- ② What is Object lock & when it is required?
- ③ While a thread executing an instance synchronized method on the given object then is it possible to execute any other synchronized method simultaneously by other threads? Ans: Not possible
- ④ What is Class Level Lock & when it is required?
- ⑤ What is the diff. b/w Object lock & class Level lock
- ⑥ What is the advantage of synchronized block over synchronized method
- ⑦ How to declare synchronized block to get class level lock?
- ⑧ What is synchronized statement? (Interview people created terminology)

- ④ The statements present in synchronized method & synchronized block are called as synchronized statement.

30/04/11

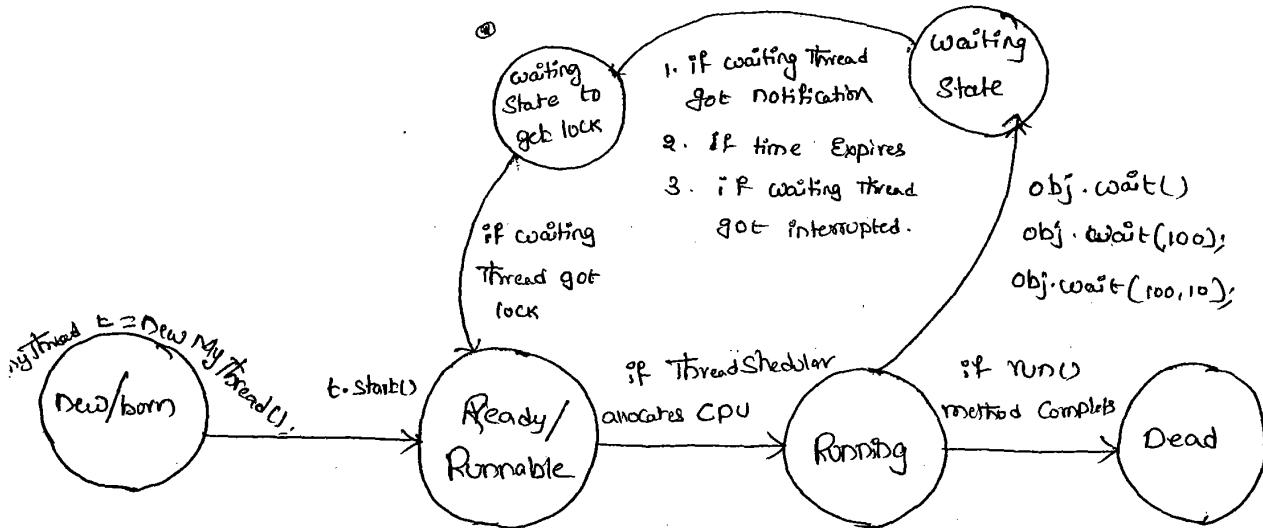
Inter Thread Communication :-

- ↳ Two threads will communicate with each other by using wait(), notify(), notifyAll() methods. The thread which requires updation it has to call wait() method. The thread which is responsible to update it has to call notify() method.
- ↳ Wait(), notify(), notifyAll() methods are available in Object class but not in Thread class. Because threads are required to call these method on any shared object.
- ↳ * If a thread wants to call wait(), notify(), & notifyAll() methods compulsorily the thread should be owner of the object. i.e., the thread has to get lock of that object. i.e., the thread should be in the synchronized area.
- Hence, we can call wait(), notify(), notifyAll() methods only from synchronized area otherwise we will get runtime exception saying "IllegalMonitorStateException".
- If a thread calls wait() method it releases the lock immediately and entered into waiting state. A thread releases the lock of only current object but not all locks. After calling notify() and notifyAll() methods thread releases the lock but may not immediately. Except these wait(), notify(), notifyAll() there is no other case where thread releases the lock.

method	is Thread releases lock?
Yield()	No
join()	No
Sleep()	No
wait()	Yes
notify()	Yes
notifyAll()	Yes

javap java.lang.Object

- 1) public final void wait() throws IE
- 2) public final native void wait(long ms) throws IE
- 3) public final void wait(long ms, int ns) throws IE
- 4) public final native void notify()
- 5) public final native void notifyAll()



Ex:-

class ThreadA

↓

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

↓

ThreadB b = new ThreadB();

b.start();

→ Thread.sleep(1000);

Synchronized (b)

↓

① System.out.println("main thread trying to call wait()");

b.wait(); // b.wait(1000);

④ System.out.println("main thread got notification");

⑤ System.out.println(b.total);

{

}

class ThreadB extends ThreadA

↓

int total = 0;

public void run()

{

Synchronized (this)

{

② System.out.println("child thread starts notification");

for(int i=1 ; i<=100 ; i++)

{

total = total + i;

{

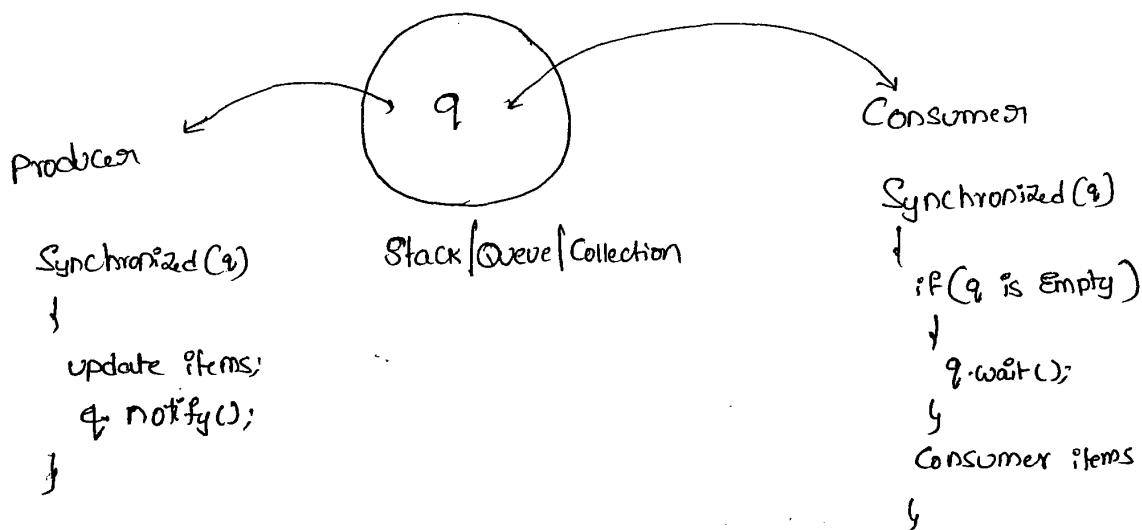
③ System.out.println("child thread trying to given notification");

{ } { this.notify(); }

O/P:- main Thread Calling wait method
 Child thread ^{stands} _{calculation}
 Child giving notification
 main Thread got notification
 5050

ctrl+c
run

Producer-Consumer problem:-



- Consumer has to Consume items from the Queue
- If Queue is Empty, he has to Call wait() method.
- producer has to produce items into the Queue.
- After producing the items, he has to Call notify() method so that all waiting Consumers will get notification.

notify() vs notifyAll():

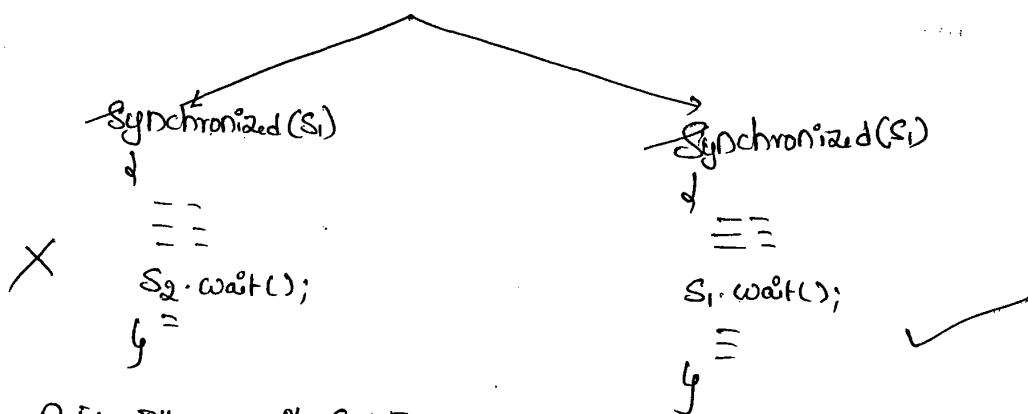
- we can use `notify()` to notify only one waiting thread. but which waiting thread will be notified we can't expect exactly. All remaining threads have to wait for further notifications.
- But in the case of `notifyAll()` all waiting threads will be notified but the threads will be executed one by one.

*Note:-

- On which object we are calling `wait()`, `notify()` & `notifyAll()`, we have to get the lock of that object.

Stack $s_1 = \text{new Stack}();$

Stack $s_2 = \text{new Stack}();$



R.E: IllegalMonitorStateException

DeadLock :-

- If two threads are waiting for each other for ever. Such type of situation is called "DeadLock".
- There are no resolution techniques for deadlock but several prevention techniques are possible.

Ex:-

Class A

↓

public synchronized void foo(B b)

↓

System.out.println("Thread1 starts execution foo");

try

↓

Thread.sleep(1000);

↳

Catch(IE e)

↓

↳

System.out.println("Thread1 trying to catch b's last()");

b.last();

↳

Public synchronized void last()

↓

System.out.println("Inside A this is last()");

↳

}

Class B

```

↓
Public synchronized void bar(A a)
{
    System.out.println("Thread2 starts bar");
    try {
        Thread.sleep(5000);
    }
    catch (InterruptedException e) {
        System.out.println("Thread 2 trying to call a's last");
        a.last();
    }
}
public synchronized void last()
{
    System.out.println("Inside B This is last");
}

```

Class DeadLock extends Thread

```

↓
A a = new A();
B b = new B();
DeadLock()
{
    this.start();
    a.foo(b); // executed by main thread
}
public void run()
{
    b.bar(a); // executed by child thread
    System.out.println("new DeadLock()");
}

```

Q.P :-

Thread1 Starts execution of foo method

Thread2 Starts execution of bar method

Thread1 trying to call b's last()

Thread2 trying to call a's last()

||

→ Synchronized keyword is the only one reason for deadlock
hence while using synchronized keyword we have to take very
much care.

* DeadLock Vs Starvation :-

→ In the case of Deadlock waiting never ends.

→ A long waiting of a thread which ends at certain point of time
is called "Starvation".

Ex :-

least priority thread has to wait until completing all the threads
but this long waiting should compulsorily ends at certain point of
time.

→ Hence, a long waiting which never ends is called "DeadLock", where
as a long waiting which ends at certain point of time is called "Starvation".

Daemon Threads :-

→ The threads which are executing in the background are called

"Daemon Threads". Ex:- Garbage Collector

→ The main objective of Daemon threads is to provide support for other non-Daemon threads.

→ We can check whether the thread is Daemon or not by using "isDaemon() method".

```
public final boolean isDaemon()
```

→ We can change Daemon nature of a thread by using setDaemon() method

```
public final void setDaemon(boolean b)
```

→ We can change Daemon nature of a thread before starting only. If we are trying to change after starting a thread we will get RuntimeException -Thread-
Saying "IllegalStateException".

→ Main thread is always Non-Daemon & it's not possible to change its Daemon nature.

Default nature :-

→ By default main thread is always non-daemon but for all the remaining threads Daemon nature will be inheriting from parent to child. i.e., if the parent is Daemon, child is also Daemon & if the parent is Non-Daemon then child is also non-Daemon.

→ whenever the last non-Daemon thread terminates all the Daemon threads will be terminated automatically.

Ex:-

Class Mythread extends Thread

{

 Public void run()

{

 For (int i=0 ; i<10 ; i++)

{

 S.o.println("Lazy thread");

 try {

 Thread.sleep(2000);

}

 Catch (InterruptedException e) { }

}

}

Catch

Class Test

{

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

{

 Mythread t = new Mythread();

 t.setDaemon(true); → ①

 t.start();

 S.o.println("end of main");

}

→ If we are Commenting Line ① Then both main & child threads are non-Daemon & hence both will be executed until their completion.

→ If we are not Commenting Line① Then main thread is non-Daemon & child thread is Daemon. Hence when ever main thread terminates automatically child thread will be terminated.

How to kill a Thread :-

→ A Thread Can Stop or Kill another Thread by using Stop() method then automatically running thread will entered into Dead State. It is a deprecated method & hence not recommended to use.

public void Stop();

Suspending & Resuming a Thread :-

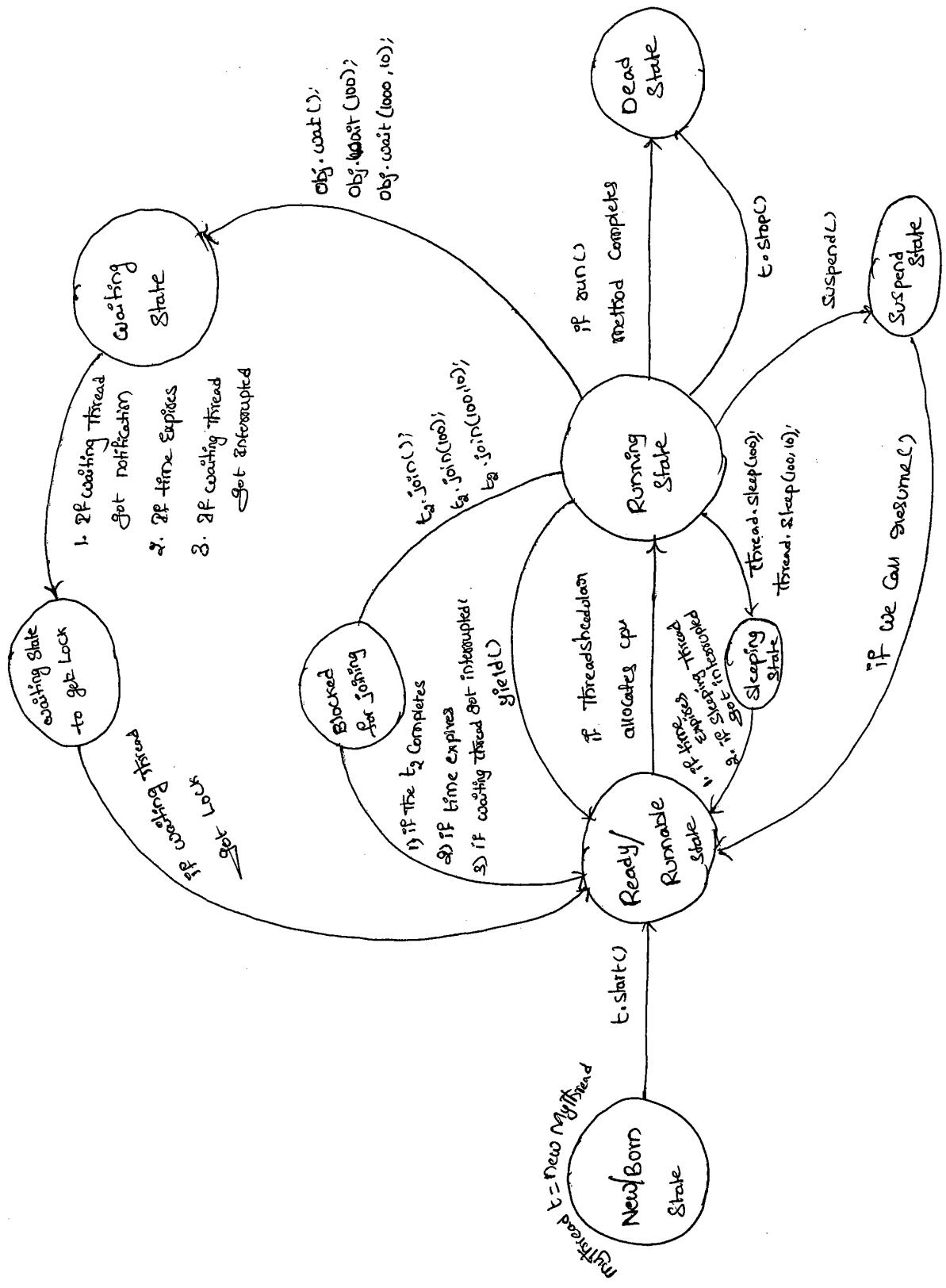
→ A Thread Can Suspend another thread by using Suspend() method.

→ A Thread Can Resume a Suspended Thread by using Resume() method.

→ Both These methods are Deprecated methods & hence not recommended to use.

Q) What is a Green Thread?

Q) What is Thread Local?



Case 4 :-

class Test

{

 public void m1(int i, float f)

{

 System.out.println("int-float version");

}

 public void m1(float f, int i)

{

 System.out.println("float-int version");

}

 P.S.V.m(____)

{

Test t₁ = new Test();

✓ t₁.m1(10, 10.5f);

✓ t₁.m1(10.5f, 10);

✗ t₁.m1(10, 10); ✗ C.E! - reference to m1() is ambiguous

✗ t₁.m1(10.5f, 10.5f); ✗ C.E! -

}

can not find symbol.

Symbol: method m1(float, float)

Location: Class Test.

Case 5 :-

Class Animal

}

↳

Class Monkey extends Animal

{

↳

Class Test

{

public void m1(Animal A)

{

S.o.println("Animal Version");

}

public void m1(Monkey m)

{

S.o.println("monkey Version");

}

P.S.V.m1()

{

Test t = new Test();

Animal a = new Animal();

t.m1(a); // Animal Version

Monkey m = new Monkey();

~~t~~.m1(m) // monkey-

Animal a1 = new Monkey();

t.m1(a1); // Animal

↳

:

→ In Overloading method resolution always takes care by Compiler based on reference type and Runtime Object never play any role in Overloading.

03/05/11

Overriding :-

- "whatever the parent has by default available to the child. If the child not satisfied with parent class implementation then child is allowed to redefine its implementation in its own way." this process is called "Overriding".
- The parent class method which is overridden is called overridden method & the child class method which is overriding is called overriding method.

Ex:- Class P

```

public void prosperity()
{
    System.out.println("Cash + Gold + Land");
}

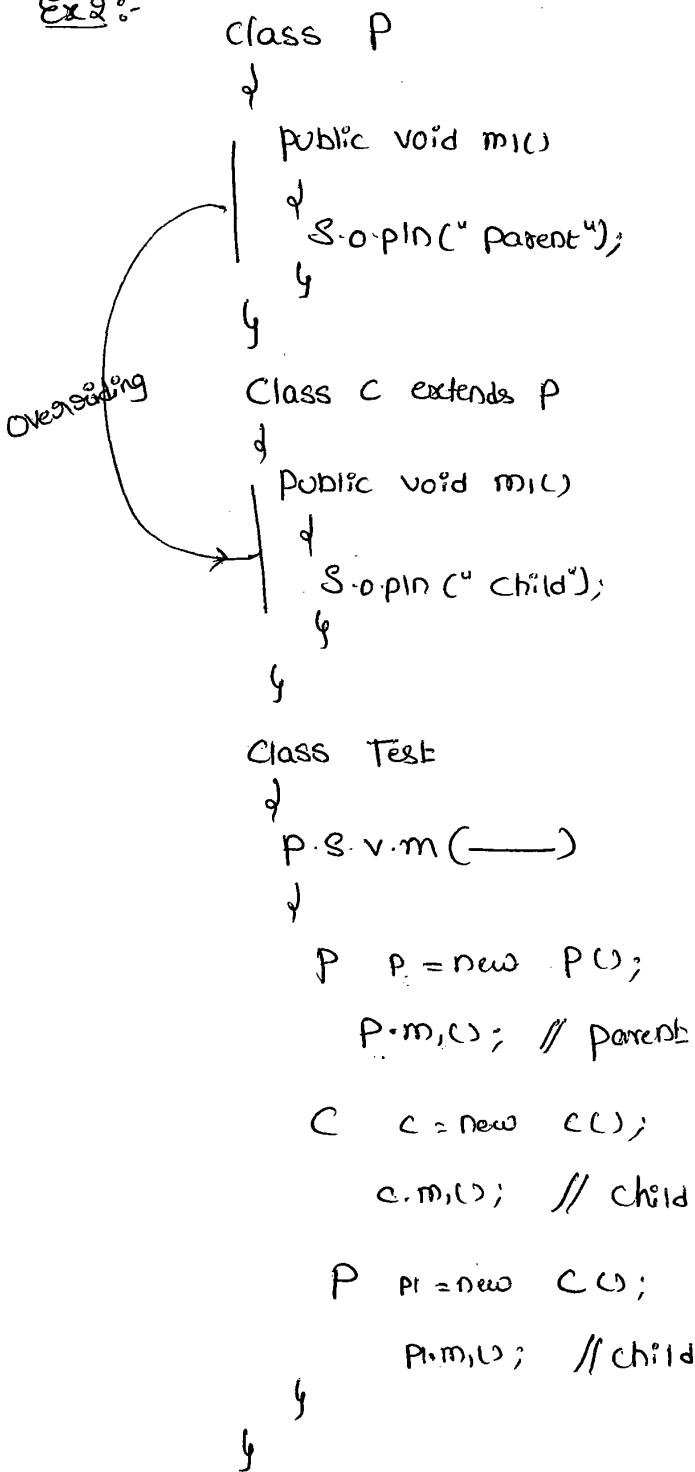
public void magisry()
{
    System.out.println("Subba Laxmi");
}

Class C extends P

public void magisry()
{
    System.out.println("Kajal | Bsha | Gtava | Hme");
}

```

Ex 2 :-



→ In overriding the method resolution always takes care by JVM based on runtime object & in overriding reference type never play any role.

212

11/04/11

Regular Expressions

→ Any group of Strings according to a particular pattern is called "Regular Expression".

Ex: ① We can write a Regular Expression to represent all valid mail-ids.

& By using that Regular Expression we can validate whether the given mail-id is valid or not.

② We can write a Regular Expression to represent all valid Java identifiers.

→ The main Application areas of Regular Expressions are:

1. We can implement validation mechanism.

2. We can develop pattern matching applications.

3. We can develop translators like Compilers, interpreters etc.

4. We can use for designing digital Circuits

5. We can use to develop Communication protocols like TCP, UDP etc.

Ex:

```
import java.util.regex.*;
```

```
class RegExDemo
```

```
{
```

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

```
{
```

```
        Pattern P = Pattern.compile("ab");
```

```
        Matcher m = P.matcher("abbabbcbdbab");
```

```

while(m.find())
{
    System.out.println(m.start() + " --- " + m.end() + " --- " + m.group());
}

```

Output :- 0 --- 2 ... ab
4 --- 6 ... ab
10 --- 12 ... ab

Pattern class:-

- A pattern object represents Compiled Version of Regular Expression.
We can Create a pattern object by using compile() of pattern class.

Pattern p = Pattern.compile(String regularExpression);

Matcher Class:-

- A Matcher object can be used to match character Sequence against a Regular Expression. We can Create a Matcher object by using Matcher() of pattern class

Matcher m = p.matcher(String target);

Important methods of matcher class:-

(i) boolean find();

- It attempts to find next match & if it is available returns True otherwise returns false.

(ii) int start();

→ returns start index of the match

(iii) int end();

→ returns end index of the match

(iv) String group();

→ returns the matched pattern

Character classes :-

① [a-z] → Any lower case alphabet symbol

② [A-Z] → Any upper case letter

③ [a-zA-Z] → Any alphabet symbol

④ [0-9] → Any digit from 0 to 9

⑤ [abc] → either a or b or c

⑥ [abc] → Except a or b or c.

⑦ [0-9a-zA-Z] → Any alpha numeric character.

Ex:-

Pattern p = Pattern.compile("x");

Matcher m = p.matcher("a3b@cuZ#");

while(m.find())

{

System.out.println(m.start() + " --- " + m.group());

}

 $x = [ab]$

0 --- a

2 --- b

 $x = [a-z]$

0 --- a

2 --- b

4 --- c

6 --- z

 $x = [0-9]$

1 --- 3

5 --- 4

7 --- 9

 $x = [0-9a-zA-Z]$

0 --- a

1 --- b

2 --- c

3 --- d

5 --- 4

6 --- 8

9 --- z

Predefined-character class :-

Space character $\longrightarrow \backslash s$
 $[0-9] \longrightarrow \backslash d$
 $[0-9a-zA-Z] \longrightarrow \backslash w$
 Any character $\longrightarrow .$

Ex:-

Pattern p = Pattern.compile ("x");

Matcher m = p.matcher ("a3z4@K7#");
 0 1 2 3 4 5 6 7 8 9

while (m.find())

{

s. o. print (m.start () + "----" + m. groups ());

}

$x = \backslash d$	$x = \backslash w$	$x = \backslash S$	$x = .$
1 ----- 3	0 --- a	5 -----	0 - a
3 ----- 4	1 --- 3		1 - 3
7 ----- 7	2 --- z		2 - z
	3 - 4		3 - 4
	6 --- k		4 - @
	7 - - - 7		5 -
			6 - 15
			7 - 7
			8 - #

Quantifiers:-

→ We can use Quantifiers to specify no. of characters to match

Ex:-

- 1) a \longrightarrow Exactly one a
- 2) a^+ \longrightarrow at least one a
- 3) a^* \longrightarrow Any no. of a 's
- 4) $a^?$ \longrightarrow atmost one a

Ex: Pattern p = Pattern.compile("a");
 Matcher m = p.matcher("abaabaaaab");
 while(m.find())
 {
 System.out.println(m.start() + "----" + m.group());
}

<u>$x=a$</u>	<u>$x=a^+$</u>	<u>$x=a^*$</u>	<u>$x=a^?$</u>
0 --- a	0 --- a	0 --- a	0 --- a
2 --- a	2 --- aa	1 -----	1 ---
3 --- a	5 --- aaa	2 ----- aa	2 --- a
5 --- a		4 -----	3 --- a
6 --- a		5 --- aaa	4 ---
7 --- a		8 ---	5 --- a
		9 ---	6 --- a
			7 --- a
			8 ---
			9 ---

Split Method (Q):

Pattern class Contains split() method → to Split given String according to a Regular Expression.

Ex: Pattern p = Pattern.compile("//");

String[] s = p.split("Durga Software Solutions");

for(String s1 : s)

{

System.out.println(s1); // Durga

Software

Solutions

Ex(9),

```
Pattern p = pattern.compile("n.");
```

```
String[] S = p.split("www.duogaojobs.com");
```

for (String s_i: s)

1

S.o.println(S1); O/P:-
www
dwangojobs
com

۲۷

String class split() method :-

→ String class also contains split() to split the given string against

A Regular Expression

Ex:-

String $s = "www \cdot designjobs \cdot com"$;

```
String[] S; = S.Split("\n");
```

for (String s₂: S)

1

S.o.plot(Sq);
www
dragajobs
com

۲۹

cam

Note 1.

Pattern class Split() can take target String as argument

where as String class split() can take regular expression as argument.

StringTokenizer :-

- We can use StringTokenizer to divide the target String into Stream of Tokens according to the
- StringTokenizer class present in java.util package.

Ex:-

① StringTokenizer st = new StringTokenizer ("Durga Software Solutions");
 while (st.hasMoreTokens())

```

  {
    System.out.println(st.nextToken());  op!-
    Durga
    Software
    Solutions
  }
  
```

Note:- The default regular Expression is Space

② StringTokenizer st = new StringTokenizer ("1,00,000", ",");
 while (st.hasMoreTokens())

```

  {
    System.out.println(st.nextToken());  op!-
    1
    00
    000
  }
  
```

QD!

```

  |
  00
  000
  
```

Ex1:- to represent
write a regular expression the set of all valid identifiers
in java language.

Rules: (1) The length of each identifier is atleast 2

(2) The allowed characters are a to z

A to Z

0 to 9

⋮

(3) The first character should not digit

R.E:- $[a-zA-Z_-][a-zA-Z0-9_-]^{*}$

$\underbrace{[a-zA-Z_-]}_x \underbrace{[a-zA-Z0-9_-]}_{x^*} \underbrace{^{*}}_{(a)}$

$x \cdot x^* = x^+$

$[a-zA-Z_-][a-zA-Z0-9_-]^+$

```
import java.util.regex.*;
```

```
class RegExDemo2
```

```
{
```

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

```
{
```

```
    Pattern p = Pattern.compile("[a-zA-Z_-][a-zA-Z0-9_-]^+");
```

```
    Matcher m = p.matcher(args[0]);
```

```
    if(m.find() && m.group().equals(args[0]))
```

```
{
```

```
        System.out.println("Valid Identifier");
```

```
}
```

```
else
```

```
{
```

```
    System.out.println("Invalid Identifier");
```

```
}
```

```
}
```

Q) W.A. RE to represent all valid mobile numbers

Note:- (1) mobile no contains 10 digits

(2) the first digit should be 7 to 9

RegEx: $[7-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]$

(or)

$[7-9]\{10\}$

Q) W.A. Regular Expression to represent all valid mail_ids

Rules:

(1) The set of allowed characters in mail_id are 0 to 9, a-z, A-Z

(2) Should starts with alphabet symbol

(3) Should contain atleast one symbol.

RegEx :-

$[a-zA-Z][a-zA-Z0-9.-]^* @ [a-zA-Z0-9]^+ ([.][a-zA-Z]^*)^+$

RegEx:-

@ gmail.com

@ (gmail | yahoo | hotmail) [.] com

Ex:-

import java.io.*;

import java.util.regex.*;

class MobileExtractor

{

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

{

PrintWriter pw = new PrintWriter("mobile.txt");

BufferedReader br = new BufferedReader(new FileReader("input.txt"));

```

String line = br.readLine();
Pattern p = Pattern.compile("[7-9][0-9]{9}");
while (line != null)
{
    Matcher m = p.matcher(line);
    while (m.find())
    {
        pw.println(m.group());
    }
    line = br.readLine();
}
pw.flush();
}

```

p) W.a.p → to Extract mail-ids from the given file where mailIds are mixed with some raw data ?

→ In the above Example Replace Regular Expression with the following mail-id Regular Expression.

$$[a-zA-Z][a-zA-Z_0-9]^* @ [a-zA-Z_0-9]^+([.][a-zA-Z]^*)^+$$

p) W.a.p → to display all text files present in the given directory ?

```

import java.io.*;
import java.util.regex.*;
class FileNameExtractor
{

```

public static void main (String[] args) throws IOException

{

int Count = 0;

Pattern p = Pattern.compile (" [a-zA-Z0-9 .] [.] txt");

File f = new File ("D:\\durga_classes");

String[] s = f.list();

for (String s1 : s)

{

Matcher m = p.matcher(s1);

If (m.find() && m.group(0).equals(s1))

{

Count ++;

s.o.println(s1);

{

s.o.println(COUNT);

{

p) W.A.P to delete all .bak files present in D:\\durga\\class

import java.io.*;

import java.util.regex.*;

class FileNamesDeletor

{

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

{

int Count = 0;

Pattern p = Pattern.compile (" [a-zA-Z0-9 .] [.] bak");

```
File f = new File("D:\\durga-classes");
String[] s = f.list();
for(String si : s)
{
    Matcher m = p.matcher(si);
    if(m.find() && m.group().equals(s1))
    {
        count++;
        System.out.println(si);
        File f1 = new File(f, si);
        f1.delete();
    }
}
System.out.println(count);
}
```

====

Enumeration (enum)

15/5/11

219
83

→ we can use enum to define a group of named Constants

Ex:- ① enum month

{

JAN, FEB, MAR ----- DEC(); → optional.

}

② enum Bear

{

KF, KO, RC, FO(); → optional

}

→ By using enum we can define our own datatypes

→ enum Concept introduced in 1.5v.

→ When compared with old languages enum Java's enum is more powerful.

Internal implementation of enum :-

→ enum Concept internally implemented by using class Concept.

→ every enum Constant is a reference variable to enum type object.

→ every enum Constant is always public static final by default.

Ex:-

enum Month

{ JAN, FEB, --- DEC; }

class month

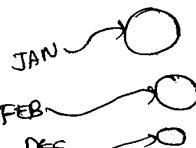
{

public static final month JAN = new Month();

public static final Month FEB = new Month();

!

Month: JAN



public static final month DEC = new Month();

}

Declaration and usage of enum :-

Ex:-

```
enum Bear
{
    KF, KO, RC, FO;
}

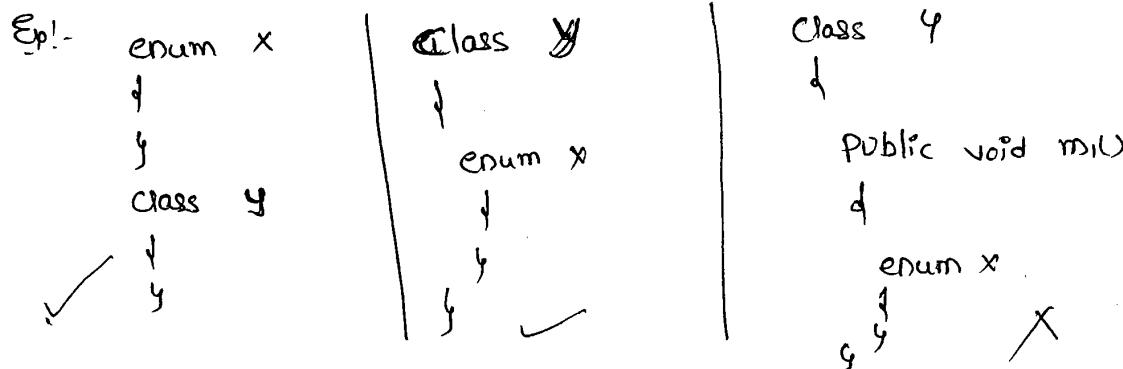
class Test
{
    p. s. v. m (Strongest args)
    {
        Bear b; = Bear.KF;
        S.o.p(b); // KF
    }
}
```

→ We can declare enum either with in the class or outside of the class but not inside a method.

→ If we are trying to declare enum with in a method we will get

Compiletime Error.

Ex! -



C.Q. - enum types must

not be local

→ if we declare enum outside the class the applicable modifiers are public, default, staticfp.

→ if we declare enum within a class the applicable modifiers are public, default, staticfp, private, protected, static.

enum Vs Switch Statement :-

→ until 1.4v the allowed datatypes for switch argument are byte, short, char, int.

→ But from 1.5v onwards in addition to above the corresponding wrapper classes ~~&~~ ^{after} Byte, Short, Char, Integer, + enum type also allowed

Swift ()	1.4v	1.5v	1.7v
	byte short char int	Byte Short Character Integer → enum	String

→ Hence from 1.5 version onwards we can use enum as argument to switch statement.

Eg:-

```

enum Beer
{
    KF, KO, RC, FO;
}

class Test
{
}

```

P . S . V . m (→)

}

Beer b₁ = Beer . R_c;

Switch (b₁)

{

Case KF :

S . o . p l n (" ZE is childean's brand");
break;

Case KO :

S . o . p l n (" ZE is too Lite");
break;

Case RC :

S . o . p l n (" ZE is challengers brand");
break;

Case FO :

S . o . p l n (" Buy one get one");
break;

default :

S . o . p l n (" other brands not recommended to take");

}

Q P . - ZE is challengers brand.

→ If we are passing enum-type as argument to Switch Statement
every Case label should be a valid enum constant.

```

Ex:- enum Beer
{
    KF, KO, RC, FO;
}

Beer b1 = Beer.KF;

switch(b1)
{
    Case KF: ✓
    Case KO: ✓
    Case RC: ✓
    Case KALYANI: X C.E.F. UnQualified Enumeration Constant name
                    required
}

```

enum Vs Inheritance :-

- Every enum in Java is direct child class of `java.lang.Enum`
- As every enum is always extending `java.lang.Enum` there is no chance of extending any other enum (because Java won't provide support for multiple inheritance).
- As every enum is always final implicitly we can't create child enum for our enums.
- Because of above reasons we can conclude inheritance concept is not applicable for enums explicitly.
- But enum can implement any no. of interfaces at a time.

Ex:- ①
enum X

↓

y

enum Y extends X

X

↓

y

C.E :-

Cannot inherit from final X

enum types not extensible

② enum X extends java.lang.Enum

↓

y

X

C.E :-

③ enum X

↓

y

X Class Y extends X

↓

y

C.E1 :- Can not inherit from final X

C.E2 :- enum types are not extensible

④ class X

↓

y

enum Y extends X

↓

y

X

C.E :-

⑤ interface ~~X~~

↓

y

enum Y implements X

↓

y



java.lang.Enum :-

- Every enum in Java ~~should~~ is always direct child class of `java.lang.Enum` class.
- The power of enum is inheriting from this class only to our enum classes.

(JavaLang:
Enum)

- It is an abstract class & direct child class of `Object` class.
- This class implements `Comparable` & `Serializable` interface. Hence every enum in java is by default `Serializable` and `Comparable`.

values() method :-

- We can use `values()` method to list out all values of enum.

Ex:- `Beer[] b = Beer.values();`ordinal() method :-

- Within the enum the order of constants is important
- we can specify its order by using `ordinal` value.
- we can find ordinal value of enum constant by using `ordinal` method.

`public int ordinal();`

- Ordinal value is zero-based.

Ex:-

enum Beer

KF, KO, RC, FO,

```

class Test
{
    p. s. v. m (String[] args)
    {
        Beeeo[] b = Beeeo. Values();
        for (Beeeo b1 : b)
        {
            S. o. pIn (b1 + " --- " + b1. ordinal());
        }
    }
}

o/p :
KF --- 0
KO --- 1
RC --- 2
FO --- 3

```

Enum class Constructors & Speciality of Java enum

- When compared with old languages enum, Java enum is more powerful because in addition to constants we can take variables, methods, constructors etc... which may not possible in old languages. This extra facility is due to internal implementation of enum concept which is class based.
- Inside enum we can declare main() method & hence we can invoke enum class directly from command prompt.

Ex:- ed. enum Fish

STAR, GOLD, GUPPY, APOLLO KILLER(), mandatory.

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

S. o. pIn ("Enum MAIN METHOD");

> Javac fish.java

> Java Fish

Q1. Enum Main method.

→ In addition to Constant if we want to take any extra members

Compulsory list of Constants should be in the 1st line & should ends with ;

e.g:- ① enum Color

↓

RED, GREEN, BLUE;

Public void m1()



② enum Color

↓

Public void m1()

↓

• RED, GREEN, BLUE;

↓



③ enum Color

↓

RED, GREEN, BLUE;

public void m1()

↓

↓



④ enum Color

↓

Public void m1()

↓

↓

X

⑤ enum Color

↓

↓

↓



→ Inside enum without having Constant we can't to take any

extra members, but Empty enum is always valid.

Eg:

enum Color

↓

Public void m1()

↓

↓

X

enum Color

↓

↓

↓



Enum Class Constructors :-

- Within Enum we can take Constructors also.
- Enum class Constructors will be executed automatically at the time of Enum class loading. Hence because Enum Constants will be created at the time of class loading only.
- We Can't invoke Enum Constructors explicitly

Ex:-

```
enum Beer
```

```
    KF, KO, RC, FO;
```

```
Beer()
```

```
    System.out.println("Constructor");
```

```
}
```

```
Class Test
```

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

```
{
```

```
    Beer b1 = Beer.KF;
```

```
    System.out.println(b1);
```

```
}
```

```
}
```

Output

```
Constructor
```

```
Constructor
```

```
Constructor
```

```
Constructor
```

```
KF
```

g2
88

→ We Can't Create Objects of Enum explicitly & hence we Can't Call Constructors directly.

Beer b = new Beer(); X

C.E! -

enum types may not be instantiated.

Ex:- enum Beer

{

KF(75), KO(90), RC(70), FO;

int price;

Beer(int price)

{

this.price = price;

}

Beer()

{

this.price = 65;

}

public int getPrice()

{

return price;

} }

class Test

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

{

Beer() b = Beer.values();

for (Beer bi : b)

{ System.out.println(bi + "----" + bi.getPrice());}

} }

KF → P.S.F. Beer KF = new Beer()
KF(100) ⇒ P.S.F. Beer KF = new Beer(100);
KF(100, "Gold", "Bitter")
⇒ P.S.F. Beer KF = new Beer(100,
"Gold", "Bitter")

O/P. KF --- 75
KO --- 90
RC --- 70
FO --- 65

- Within the enum we can take instance & static methods but we can't to take abstract methods
- every enum constant represents an object hence whatever ever the methods we can apply on ~~enum~~^{Normal Java} object we can apply those on enum constants also.

Ex:-

- Q) which of the following expressions are valid
- ✓ ① Beer1.KF.equals(Beer1.RC) // True
 - ✓ ② Beer1.KF.hashCode() // ✓
 - ✓ ③ Beer1.KF == Beer1.RC → false
 - X ④ Beer1.KF > Beer1.RC
 - ✓ ⑤ Beer1.KF.ordinal() > Beer1.RC.ordinal

Case(1):-

```
Package Pack1;
public enum Fish{
    STAR, GUPPY, APOLO;
}
```

```
Package Pack2;
class Test1
{
    p.s.v.m();
    System.out.println(STAR);
}

import static Pack1.Fish.STAR;
(a)
import static Pack1.Fish.*;
```

28/1

Package pack3;

--
Class Test2

↓

P.S.V.M(→)

↓

Fish f = Fish.STAR;

S.O.Pln(f);

↓

import pack1.Fish;

(or)

import pack1.*;

package pack4;

--
Class Test3

↓

P.S.V.M(→)

↓

Fish f = Fish.STAR;

S.O.Pln(Guppy);

↓

import pack1.Fish (or)

import pack1.*;

import static pack1.Fish.Guppy;

(or)

import static pack1.Fish.*;

Case 2 :-

enum Color

↓

BLUE, RED

↓

public void info()

↓

S.O.Pln("Dangerous Color");

↓, GREEN;

public void info()

↓

S.O.Pln("Universal Color");

↓

```

class Test
{
    public static void main()
    {
        Color[] c = Color.values();
        for (Color c1 : c)
        {
            c1.info();
        }
    }
}

```

Op :-
 Universal Color
 Dangerous Color
 universal Color.

Enum vs Enum vs Enumeration :-

enum :-

→ It is a Keyword which can be used to define a group of named Constants.

Enum :-

→ It is a class present in java.lang package which acts as a base class for all Java enums

Enumeration :-

→ It is an Interface present in java.util package, which can be used for retrieving objects from Collection one by one.

Internationalization (I18N)

I18N :-

→ Various Countries follow various Conventions to represent dates & No's etc.

→ Our application should generate Locale Specific responses like for India People the response should be in terms of Rs. (Rupees) & for the US people the response should be in terms of dollars. The process of designing such type of web application is called "Internationalization" (I18N).

→ We can implement I18N by using the following classes

- ① Locale
- ② NumberFormat
- ③ Dateformat

Locale :-

→ A Locale Object represents a Geo-graphic Location

Constructors :-

→ We can Create a Locale object by using the following Constructor.

- (1) `Locale l = new Locale(String language);` javap `javutil.Locale`
- (2) `Locale l = new Locale(String language, String Country);`

→ Locale Class defines Several Constants to represent Some Standard

Locales. we can use these Locale directly without Creating our own.

- | | | |
|------|------------------------|-----------------------------|
| Ex:- | <code>Locale.US</code> | <code>Locale.ITALIAN</code> |
|------|------------------------|-----------------------------|

- | | |
|-----------------------------|------------------------|
| <code>Locale.ENGLISH</code> | <code>Locale.UK</code> |
|-----------------------------|------------------------|

Note:-

- Locale class is the final class present in java.util package
- It is the direct child class of Object it implements Cloneable & Serializable interfaces.

Important methods of Locale class:-

- ① Public static Locale getDefault();
- ② public static void setDefault(Locale l);
- ③ public String getLanguage(); en
- ④ public String getDisplayLanguage(); english
- ⑤ public String getCountry(); us
- ⑥ public String getDisplayCountry(); unitedstates
- ⑦ public static String[] getISOCountries();
- ⑧ public static String[] getISOLanguages();
- ⑨ public static Locale[] getAvailableLocales();

Ex:-

```
import java.util.*;  
  
class LocaleDemo1  
{  
    public static void main(String[] args)  
    {  
        Locale l1 = Locale.getDefault();  
        System.out.println(l1.getCountry() + " --- " + l1.getLanguage());  
        System.out.println(l1.getDisplayCountry() + " --- " + l1.getDisplayLanguage());  
  
        Locale l2 = new Locale("pa", "IN");  
        Locale.setDefault(l2);  
  
        String[] s3 = Locale.getISOLanguages();  
        for (String s4 : s3)  
        {  
            System.out.println(s4);  
        }  
  
        String[] s4 = Locale.getISOCountries();  
        for (String s5 : s4)  
        {  
            System.out.println(s5);  
        }  
  
        Locale[] s = Locale.getAvailableLocales();  
        for (Locale s1 : s)  
        {  
            System.out.println(s1.getDisplayCountry() + " --- " + s1.getDisplayLanguage());  
        }  
    }  
}
```

NumberFormat Classes :-

→ Various Countries follow various Conventions to represent Number by using NumberFormat Class we can format a number according to a particular Locale.

→ NumberFormat class present in java.text package & it is an abstract class. Hence we can't Create NumberFormat Object directly

X NumberFormat nf = new NumberFormat();

Creating NumberFormat object for the default Locale :-

→ NumberFormat class defines the following methods for this

- ① public static NumberFormat getInstance();
- ② public static NumberFormat getCurrencyInstance();
- ③ public static NumberFormat getPercentInstance();
- ④ public static NumberFormat getNumberInstance();

Getting NumberFormat object for a Specific Locale :-

→ We have to pass the corresponding Locale object as argument to the above methods

- Ex:-
- ① public static NumberFormat getCurrencyInstance(Locale l);
!

→ Once we got NumberFormat Object we can format & parse numbers by using the following methods of NumberFormat class

- ① public String format(long l);
- ② public String format(double d);

→ To format or Convert java Specific number form to Locale Specific String form.

- ③ Public Number parse(String s) throws ParseException

→ To Convert Locale Specific String form to java Specific Number form.

Ex:-

WAP to represent a Java number in Italy Specific form.

① import java.text.*;

import java.util.*;

Class NumberFormatDemo2

{

P. S. v. m();

{

double d = 123456.789;

NumberFormat nf = NumberFormat.getInstance(Locale.ITALY);

S. o. println("Italy form is: " + nf.format(d));

}

Output: Italy form is: 123.456,789

Expt. - Q. a p to represent a java number in India, UK &

U.S Currency forms.

```
import java.text.*;
```

```
import java.util.*;
```

```
Class NumberFormatDemo3
```

```
{
```

```
    P.S.V.M( — )
```

```
}
```

```
double d = 123456.789;
```

```
Locale india = new Locale("pa", "IN");
```

```
NumberFormat nf1 = NumberFormat.getCurrencyInstance(india);
```

```
S.o.println("India notation is...." + nf1.format(d));
```

```
NumberFormat nf2 = NumberFormat.getCurrencyInstance(Locale.US);
```

```
S.o.println("US notation is...." + nf2.format(d));
```

```
NumberFormat nf3 = NumberFormat.getCurrencyInstance(Locale.UK);
```

```
S.o.println("UK notation is...." + nf3.format(d));
```

```
}
```

```
.
```

O/p:- India Notation is INR 123,456.79

US Notation is \$ 123,456.79

UK Notation is £ 123,456.79

Setting maximum & minimum integer & fraction digits:

→ NumberFormat class defines the following methods to set maximum & minimum fraction & integer digits.

- ① Public void SetMaximumFractionDigits (int n);
- ② Public void SetMinimumFractionDigits (int n);
- ③ Public void SetMaximumIntegerDigits (int n);
- ④ Public void SetMinimumIntegerDigits (int n);

18/5/11

Ex:-

- ```
NF nf = NF.getInstance();
① nf.setMaximumFractionDigits(2);
 S.o.println(nf.format(123.4567)); // 123.45
 S.o.println(nf.format(123.4)); // 123.4
② nf.setMinimumFractionDigits(2);
 S.o.println(nf.format(123.4567)); // 123.4567
 S.o.println(nf.format(123.4)); // 123.40
③ nf.setMaximumIntegerDigits(2);
 S.o.println(nf.format(123456.234)); // 123456.234
 S.o.println(nf.format(12.3456)); // 12.3456
④ nf.setMinimumIntegerDigits(3);
 S.o.println(nf.format(123456.234)); // 0123456.234
 S.o.println(nf.format(12.3456)); // 00012.3456
```

## Dateformat class :-

- Various Countries follow various Conventions to represent Date.
- By using DateFormatt class we can format the DATE according to a particular Locale.
- DateFormat class is an abstract class & present in java.text package.

## Getting DateFormat Object for Default Locale :-

DateFormat class defines the following methods for this

- ① public static DateFormat getInstance();
  - ② public static DateFormat getDateInstance();
  - ③ public static DateFormat getDateInstance(int Style);
- DateFormat · Full → 0  
DateFormat · LONG → 1  
DateFormat · MEDIUM → 2  
DateFormat · SHORT → 3

## Getting DateFormat object for the Specific Locale :-

- ① Public static DateFormat getDateInstance(int style, Locale l);

→ Once we got DateFormat Object we can format & parse dates by using the following methods -

- ① Public String format( Date d);

→ To Convert Java Date form to Locale Specific String form

- ② Public Date parse(String s) throws ParseException

To Convert Locale Specific Date Form to java DateForm.

Ex:-

W-a-P To display System Date in all possible Styles of U.S format

```
import java.util.*;
```

```
import java.text.*;
```

```
Class DateFormatDemo
```

```
{ P. S. V. m (_____) }
```

```
) {
S. o. p(" full form :" + DateFormat . getDateInstance(0).
format(new Date()));
```

(or)

```
// DateFormat df = DateFormat . getDateInstance(0);
```

```
S. o. p(df. format(new Date()));
```

```
S. o. p(" Long form :" + DF . getDateInstance(0) . format(new Date));
```

```
S. o. p(" medium form :" + DF . getDateInstance(2) . format(new Date));
```

```
S. o. p(" short form :" + DF . getDateInstance(3) . format(new Date));
```

```
}
```

QD:- full form: Thursday, February, 2, 2010

Long form: February 18, 2010

medium form: Feb 18, 2010

short form: 2/18/10

Ex 2).

① w.a.p to display System Date US, UK & Italy form.

{

```
S.o.println("US form:" + DF.getDateInstance(0, Locale.US).format(new Date()));
```

```
S.o.println("UK form:" + DF.getDateInstance(0, Locale.UK).format(new Date()));
```

```
S.o.println("ITALY form:" + DF.getDateInstance(0, Locale.ITALY).format(new Date()));
```

}

Op.

US form : Tuesday, May 18, 2010

UK form : Tuesday, 18 May 2010

ITALY form: martedì 18 maggio 2010

Getting DateFormat object to represent both DATE & TIME:

① public static DateFormat getDateTimeInstance();

② public static DateFormat getDateTimeInstance(int datestyle, int timestyle);

③ public static DateFormat getDateTimeInstance(int datestyle, int timestyle, Locale);

Ex).

```
S.o.println("US form :" + DateFormat.getDateTimeInstance(0, 0, Locale.US)
 .format(new Date()));
```

Op.

US form: Tuesday, May 18, 2011 9:53:45 AM GMT +5:30

Note:- Default style is medium & most of the cases default Locale is US

22/95

Note:-

Default Style is Medium & most of the Cases default Locale is  
US



## Development

23/96

### javac :-

We can use this Command to Compile a Single or group of .java files.

Syn:-

```
javac [options] A.java /
 ↓
 -d A.java B.java
 -source
 -cp
 -classpath
 -version
```

### java :-

We can use java Command to run a .class file

Syn:-

```
java [options] A ←
 ↓
 -ea | -esa | -da | -dsa
 -version
 -cp / -classpath
 -D
```

Note:- We can compile a group of .java files at a time whereas we can run only on .class file at a time.

### Classpath :-

→ Classpath describes the location where required .class files are available.

→ JVM will always use Classpath to locate the required .class file.

→ The following are various possible ways to Set the Classpath.

① permanently by using Environment variable classpath.

→ This classpath will be preserved after system restart also

② At Command prompt level by using Set Command.

Set classpath = %classpath% ; D:\path >

→ This classpath will be applicable only for that particular Command prompt window only. Once we close that Command prompt automatically classpath will be lost.

③ At Command Level by using -cp option

java -cp D:\path > Test ↴

→ This classpath is applicable only for this particular Command.

Once Command execution completes automatically classpath will be lost.

\* Among the above 3 ways the most commonly used approach is  
Setting classpath at Command Level.

Ex:- Class Test

```
↓
p. S. v. m (—)
↓
S. o. p. n (" classpath Demo");
↳
```

D:\DurgaClasses > javac Test.java ↴

> java Test ↴

Op:- Classpath Demo

X D:\java Test ↴ R.E!- NoClassDefFoundError

✓ D:\> java -cp D:\DurgaClasses Test ↴ ✓  
Op:- classpathDemo

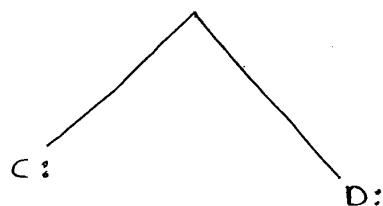
✓ D:\> java -cp D:\DurgaClasses Test ↴  
Op:- classpathDemo

97

Note!.

If we set classpath explicitly then we can run Java program from any location but if we are not setting the classpath then we have to run java program only from current working directory.

Ex 2 :-



```
public class fresher
{
 public void m1()
 {
 System.out.println("I want job");
 }
}
```

Class Company  
↓  
P.S.V.M (→)

fresher = new fresher();

$f_m(u)$

S.O.Plan ("Getting JOB is very  
easy .. not required to  
worry");

C:\>javac foreshort.java ✓

D:\> javac Company.java

C.E:- Cannot find Symbol

Symbols: class freshly

location : Class Company

D:\>javac -cp c: Company.java

X D:\java& Company ↵

## R.E.: NoClassDefFoundError: fresher

~~x D:\java -cp c: Company~~

R.E:- NoClassDefFoundError: Company

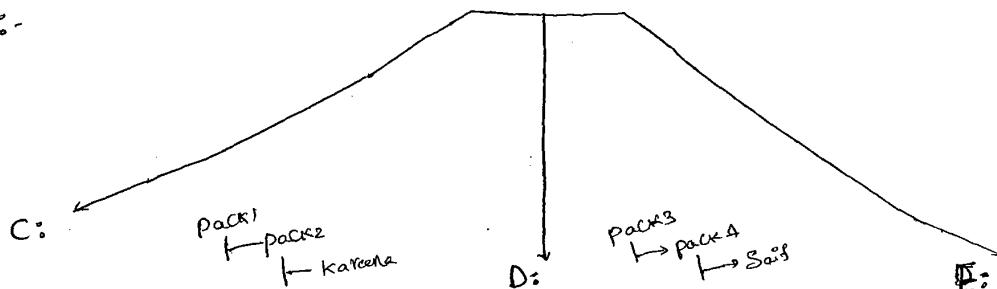
✓ D:\> java -cp D:\;C:\ Company      (or) D:\> java -cp .;C:\ Company

O/P:- I won Job

Getting Job is very easy... not required to worry.

✓ E:\> java -cp D:\;C:\ Company

Ex:-



Package pack1.pack2;

public class Kareena

{

    public void m1()

}

S.o.println("Hello Saif Can u

    please setz hello  
    ---funz");

    }

    }

package pack3.pack4;

import pack1.pack2.Kareena

public class Saif

{

    public void m1()

{

    Kareena k = new Kareena();

    k.m1();

    S.o.println("Not possible.. As I am

    in static class");

{

import pack3.pack4  
    .Saif;

class Durga

{

    P.S.V.m1();

{

    Saif s = new Saif();

{

    s.m1();

{

    S.o.println("Can

{

    I help U");

{

✓ C:\> java -d . Kareena

✗ D:\> java -d . Saif.java

C:\> - Cannot find Symbol

Symbol: class Kareena

Location: class Saif

Ques 2  
98

✓ D:\>java -cp c: -d . Saif.java

✗ E:\>javac Durga.java

C.E: cannot find Symbol

Symbol: class Saif

Location: class Durga

✓ E:\>javac -cp D: Durga.java

✗ E:\>java Durga ←

R.E: NoClassDefFoundError : Saif

✗ E:\>java -cp D: Durga ←

R.E: NoClassDefFoundError : Durga

✗ E:\>java -cp .;D: Durga

R.E: NoClassDefFoundError : ~~Durga~~ Durga

✓ E:\>java -cp E:;D;;c: Durga

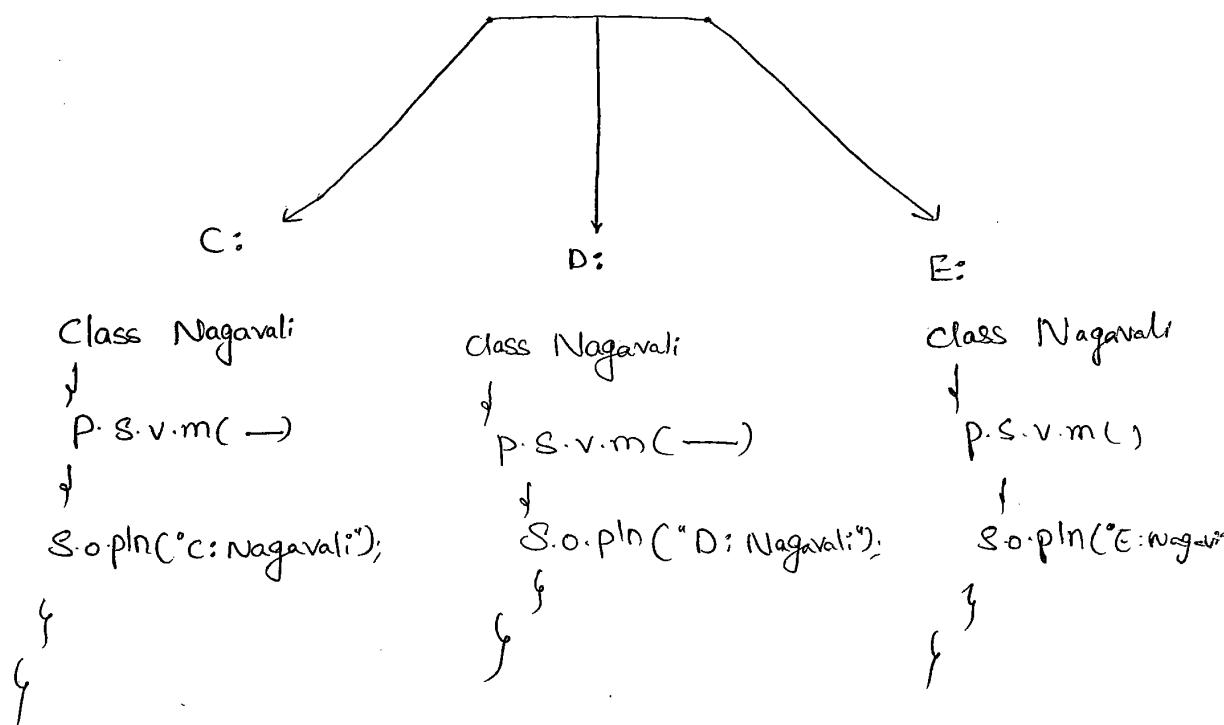
Note:

① Compiler will check only one level dependency whereas JVM will check all levels of dependency

② If any folder structure created because of package statement it should be mentioned through import statement only. If Base package location we have to update in classpath.

③ Within the classpath the order of locations is very important for the required .class file, JVM will always search the locations from

Left → Right in classpath. Once JVM finds the required .file then  
The rest of the classpath won't be searched.



C:\> javac Nagavali.java ✓

D:\> javac Nagavali.java ✓

E:\> javac Nagavali.java ✓

C:\> java Nagavali ✓

o/p C: Nagavali

D:\> java -cp C:; D:; E: Nagavali ←

o/p C: Nagavali

D:\> java -cp E:; D:; C: Nagavali ←

o/p!- E: Nagavali

D:\> java -cp D:; E:; C: Nagavali ←

o/p! D: Nagavali

## JAR file :-

237 99

→ If several dependent files are available then it is never recommended to set each class file individually in the classpath we have to group all those .Class file into a single Zip file. & we have to make that Zip file available in the classpath. This zip file is nothing but JAR file.

## Ex:-

To develop Servlet all required .class files are available in Servlet-api.jar. we have to make this jar file available in the classpath then only Servlet will be compiled.

## Jar vs War vs EAR :-

### ① Jar :- (Java archive file)

→ It Contains a group of .class files

### ② War :- (Web archive file)

→ It represents a web application which may contain Servlets, JSPs, HTMLs, CSS file, JavaScripts, etc..

### ③ EAR :- (Enterprise archive file)

→ It represents an enterprise application which may contains Servlets, JSPs, EJBs, JMS Components etc..

## Various Commands :-

- ① To Create a jar file.

```
jar -cvf durga.jar A.class B.class C.class
 *.class
```

- ② To extract a jar file.

```
jar -xvf durga.jar
```

- ③ To Display table of contents of a jar file.

```
jar -tvf durga.jar
```

Ex:-

```
public class DurgaColorfullCalc
{
 public static int add(int x, int y)
 {
 return x*y;
 }
 public static int add(int x, int y)
 {
 return 2*x*y;
 }
}
```

C:\> javac DurgaColorfullCalc.java ✓

C:\> jar -cvf durgacalc.jar DurgaColorfullCalc.class

23/ 100

```

class Bakaria
{
 public static void main()
 {
 System.out.println(DusgaColorfulCalc.add(10, 20));
 System.out.println(DusgaColorfulCalc.multiply(10, 20));
 }
}

```

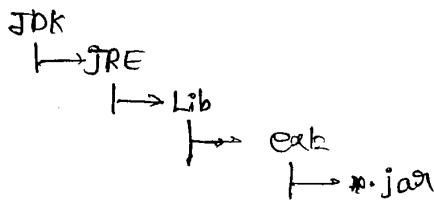
- ✗ D:\> javac Bakaria.java
- ✗ D:\> javac -cp C: Bakaria.java
- ✓ D:\> javac -cp C:\dusgacalc.jar Bakaria.java
- ✓ D:\> javac -cp ;C:\dusgacalc.jar Bakaria.java
- Op:- 200  
400

#### Note :-

- whenever we are placing a jar file in the classpath
- Compulsory name of the jar file we should include, Just Location is not enough.

#### Shortcut way to place jar file :-

- If we are placing the jar file in the following location then it is not required to set classpath explicitly by default if it is available to Jvm & Java Compiler.



## System properties :-

- For every System persistence information will be maintain in the form of System properties. These may include O.S name, Hardware machine version, User Country - e.t.c....
- we can get System properties by using `getProperties()` method of System class

Ex:- Demo program to print all System properties.

```
import java.util.*;

class Test
{
 public static void main (String[] args)
 {
 Properties p = System.getProperties();
 p.list(System.out);
 }
}
```

- we can set System property from the Command prompt by using `-D` option

ex:- Java -D<sup>Space is not allowed</sup>  
duong=SCJP Test

↓  
name of the property

↓  
value of the property

## Q) JDK vs JRE vs JVM :-

93<sup>b</sup> 101

### JDK : (Java development kit) :-

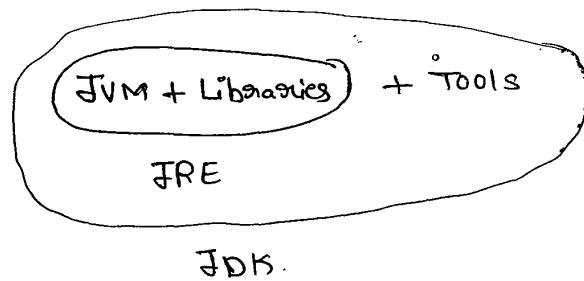
→ To develop & run Java application the required environment provided by JDK.

### JRE :- (Java Runtime Environment) :-

→ To run Java application the required environment provided by JRE

### JVM :-

→ This machine is responsible to execute Java program.



$$\text{JDK} = \text{JRE} + \text{Tools}$$

$$\text{JRE} = \text{JVM} + \text{Libraries}.$$

### Note:-

→ On Client machine we have to install JRE, whereas on The developer's machine we have to install JDK.

## diff. b/w path & classpath :-

- we can use classpath to describe the location where required class files are available.
- If we are not setting the classpath then our program won't be run.

## Path :-

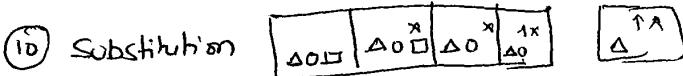
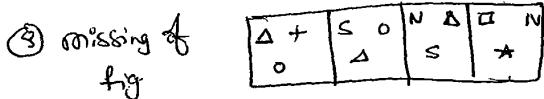
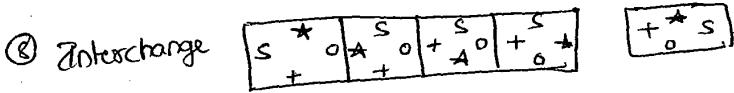
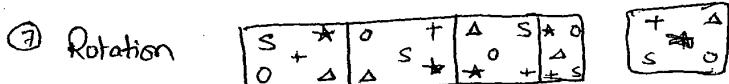
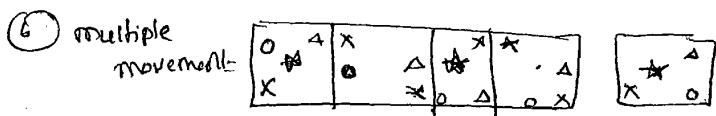
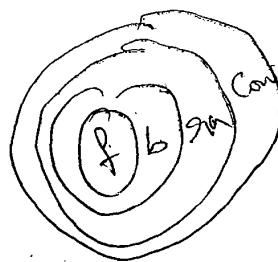
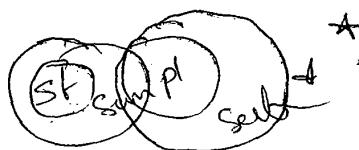
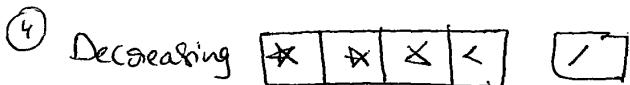
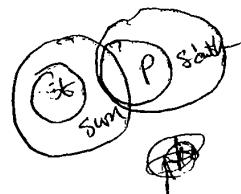
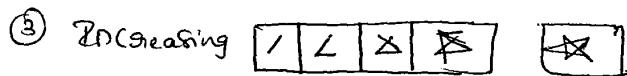
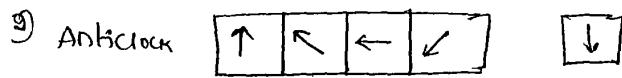
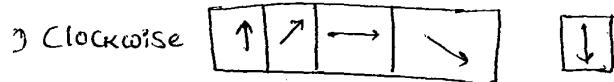
- we can use path variable to describe the location where required Binary executables are available.
- If we are not setting path variable then java & javac Commands won't work.

236

卷之三

232

1  
2  
3



238

卷之三

miracle

- 1.5V → Autoboxing & Unboxing -  
→ generics;  
→ varargs -  
→ for-each  
→ enum  
→ Annotations -  
→ Queue ✓  
→ static imports / not recommended ✓  
→ Co-Varic of return types.

Walk  
↓  
Jogging  
↓  
Running  
↓  
Sprinting

Siddhartha (VNR VJIT)  
9951884313  
Siddhartha88@yahoo.co.in

Vishnuteja.Y.S  
9703346473, 9495410648

Vishnuteja87@gmail.com

Vasu - 8779969444 (EVNG)

Slvud Narma@gmail.com.

26/02/11

239

## Garbage Collection

- 1) Introduction.
- 2) Various ways to make an object eligible for G.C.
- 3) The methods for requesting JVM to run garbage collector.
- 4) Finalization..

### Garbage Collector :-

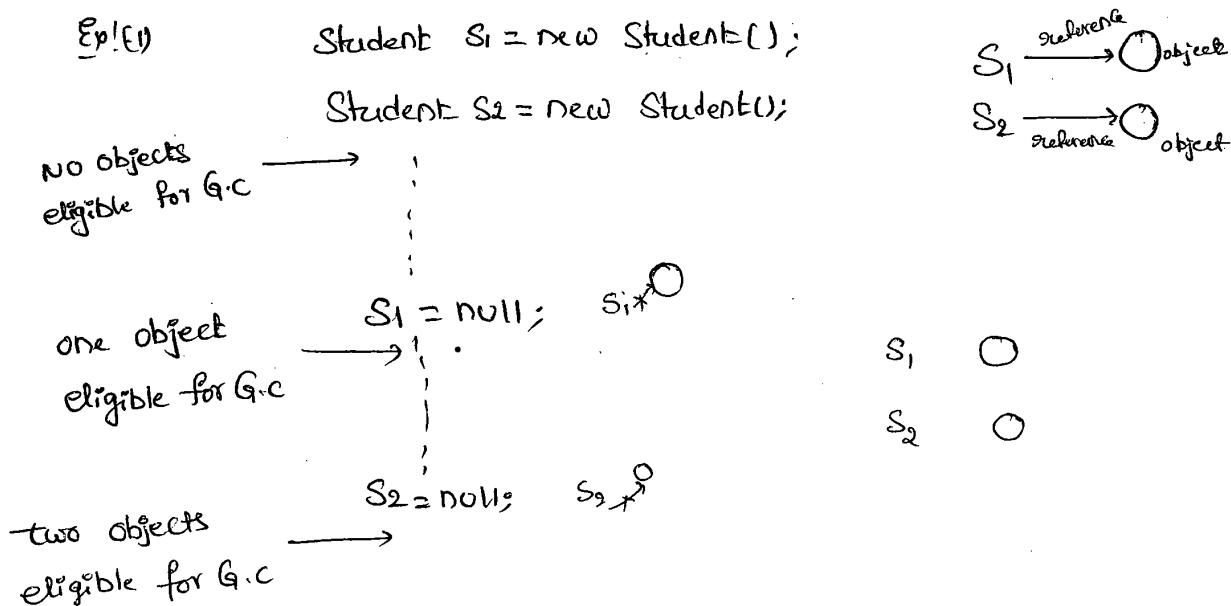
- In old languages like C++, creation & destruction of object is responsibility of programmer only.
- Usually programmer taking very much care while creating objects & his neglecting destruction of useless objects. due to this neglectance at <sup>Certain</sup> second point of time for the creation of new object sufficient memory may not be available & entire program will be collapsed due to memory problems.
- But in Java, programmer is responsible only for creation of objects and he is not responsible for destruction of useless objects.
- Sun people provided one assistant which is always running in the background for destruction of useless objects. due to this assistant the chance of failure java program with memory problem is very little. This assistant is nothing "Garbage Collector".
- Hence, the main objective of Garbage Collector is to "destroy useless objects".

## The Various ways to make an object eligible for G.C:-

- Even though programmer is not responsible to destroy useless object, it is always a good programming practice to make an object eligible for G.C if it is no longer required.
- An object is said to be eligible for G.C, if it doesn't contain any references.
- The following are various possible ways to make an object eligible for G.C.

### (i) Nullifying the reference Variable :-

- If an object is no longer required then assign null to all its references, then automatically that object eligible for G.C.

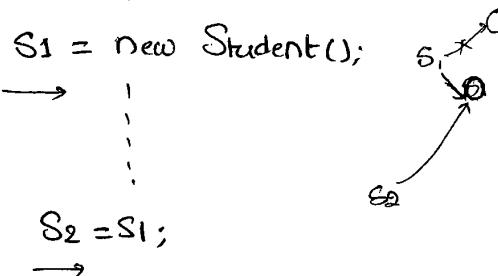
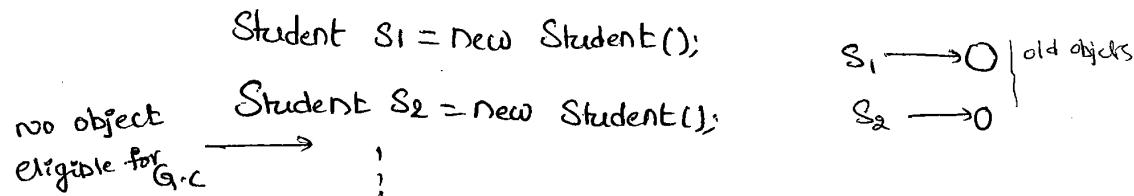


## 2) Reassigning the Reference Variable:

240

→ If an object is no longer required then reassign its reference variables to some other objects then that old object automatically eligible for G.C.

Ex:-



## 3) Objects Created Inside a method :-

→ The Objects which are created inside a method are by default eligible for G.C after completing that method.

Ex:- Class Test

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

2 objects eligible for G.C.

m<sub>1</sub>();

p.s.v.m<sub>1</sub>()

Student S<sub>1</sub> = new Student();  
Student S<sub>2</sub> = new Student();

Ex 2 :-

Class Test

↓  
p.s.v.m(String[] args)

}

one object  
eligible for  
G.C

Student s = m();

}

p.s.Student m()

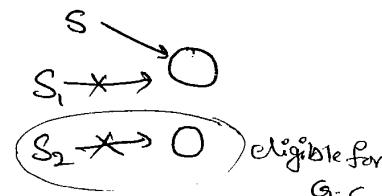
{

Student s1 = new Student();

Student s2 = new Student();

return s1;

}



$s_1 \rightarrow \textcircled{O}$

$s_2 \rightarrow \textcircled{O}$

$s \rightarrow \textcircled{O}$

$s_1 \rightarrow \textcircled{O}$

Ex 3 :-

Class Test

↓

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

}

Two objects  
eligible for  
G.C

m();

}

p.s.Student m()

{

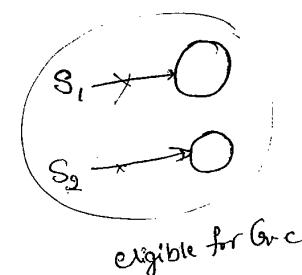
Student s1 = new Student();

Student s2 = new Student();

return s1;

}

}



## 4) Island of isolation :-

241

Ex:- Class Test

Test i;

P.S.V. main(String args)

{

Test t1 = new Test();

Test t2 = new Test();

Test t3 = new Test();

No objects  
eligible for  
G.C

t1.i = t2;

t2.i = t3;

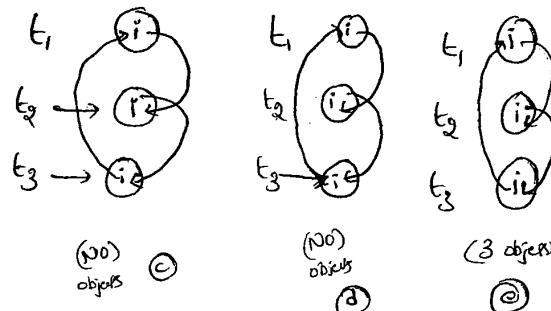
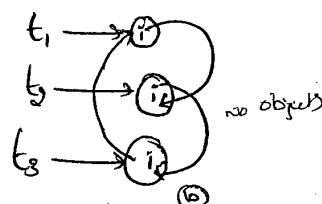
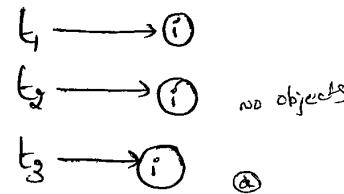
t3.i = t1;

t1 = null;

t2 = null;

t3 = null;

3 objects  
eligible for  
G.C



### Note:-

- If an Object doesn't have any Reference Then it is always eligible for Garbage Collector.
- Even though object having ~~the~~ the Reference still it is eligible for G.C Sometimes (island of isolation)

## The methods for requesting JVM to Run Garbage Collector:-

- Whenever we are making an object eligible for G.C it may not be destroyed by G.C immediately whenever JVM runs garbage collector then only that object will be destroyed.
- We can request JVM to run garbage collector, programmatically whether JVM accepts our request are not there is no guarantee.
- The following are various ways for this requesting JVM to run G.C.

### (1) By System class :-

- System class contains a static method G.C, for this  
`System.gc();`

### (2) By Runtime class :-

- By using Runtime object a Java application can communicate with JVM
- Runtime class is a Singleton class hence we can't create Runtime object by using constructor.
- we can create a Runtime object by using factory method `getRuntime()`  
`Runtime r = Runtime.getRuntime();`
- Once we got Runtime object we can apply the following methods on that object.
  - a) `freeMemory()` returns free memory in the heap,
  - b) `totalMemory()` → total a of the Heap (`HeapSize`)
  - c) `Gc()` → for requesting JVM to Run garbage collector.

exp:- class RuntimeDemo

247

{  
    p.s.v.main(String[] args)  
}

Runtime r = Runtime.getRuntime();

S.o.println(r.totalMemory());

S.o.println(r.freeMemory());

for (int i=1; i<=10000; i++)

{

Date d = new Date();



d=null;



S.o.println(r.freeMemory());

r.gc();

System.out.println(r.freeMemory());

if

Q) Which of the following is the proper way of requested JVM to run GC.

1) System.gc(); (System is static method)

2) Runtime.gc(); (Runtime is instance method)

3) (new Runtime()).gc(); (GC is applicable only static method)

4) Runtime.getRuntime().gc();

Note:- gc present in the System class is a static method, whereas

gc present in the Runtime class is instance method & recommended to

use System.gc();

## Finalization :-

- Just before destroying any object, garbage collector always calls finalize() method to perform clean-up activities on that object.
- finalize() method declared in Object class with the following declaration.

protected void finalize() throws Throwable.

### Case(1) :-

- Garbage Collector always calls finalize() on the object which is eligible for G.C just before destruction, then the corresponding class finalize() will be executed. If String object eligible for G.C then String class finalize() will be executed. but not Test class finalize method.

Ex:- class Test

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

 String s = new String("durga");
 s = null;
 System.gc();
 System.out.println("end of main");

 public void finalize()
 {
 System.out.println("finalize method called");
 }
}
```

O/p :- end of main

→ In the above Example String object is eligible for g.c. Hence  
String class finalize() method got executed which has Empty implementation. 243

→ If we are replacing String object with Test object, Then Test class  
finalize() will be executed.

→ In this case the o/p is ① finalize method called

End of main (a)

② end of main

finalize method called.

### Case 2 :-

→ We can Call finalize() Explicitly in this case it will be executed

→ Just like a normal method call & Object won't be destroyed.

→ But Before destruction of object G.C always Call finalize().

Ex. Class Test

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

Test t = new Test();

t.finalize();

t.finalize();

t=null;

System.gc();

S.o.pn("end of main");

o/p.

finalize method called

finalize method called

end of main

finalize method called

Public void finalize()

S.o.pn("finalize method called");

→ In the above program finalize() got executed 3 times, 2 times explicitly by the programmer & one time by the Garbage Collector.

Note :-

- Before destruction of Servlet object Web Container always calls destroy method, to perform clean-up activities. But
- It is possible to call destroy() explicitly from init() & service(). In this case it will be executed just like a normal method call and Servlet object won't be destroyed.

Case(3) :-

- If we are calling finalize() explicitly & while executing that finalize(), if any exception raised & uncaught, then the program will be terminated abnormally.
- If G.C calls finalize() & while executing that finalize(), if any exception raised is uncaught no corresponding catch block then Jvm simply ignores that uncaught exception & rest of the program will be executed normally.

Ex:- class Test

```
 {
 p.s.v.m (String [] args)
 }
 Test t = new Test();
 t.finalize(); → Line①
 t = null;
 System.gc();
 S.o.pIn("end of main");
}
```

```

public void finalize()
{
 System.out.println("finalize method called");
 System.out.println(100);
}

```

→ If we are not Comment Line①, then we are Calling the `finalize()` Explicitly and the program will be terminated abnormally.

→ If we are Commenting Line①, then G.C calls `finalize()` & The raised A.E is ignored by JVM. Hence in this Case the o/p is  
O/P: end of main!  
finalize method Called.

Q) Which of the following Statement is True?

X) While executing `finalize()` all exceptions are ignored by JVM.

✓) While " " only uncaught exceptions ignored by JVM.  
no caught block

Conclusion:

→ on any object G.C calls `finalize()` only once.

Note!

→ The Behaviour of G.C is vendor dependent & hence we can't expect explicitly because of this we can't answer]

Ex Class finalizeDemo

{

    Static finalizeDemo s;

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

{

        finalizeDemo f = new finalizeDemo();

        S.o.println(f.hashCode());

        f=null;

        System.gc();

        Thread.sleep(5000);

        System.out.println(s.hashCode());

        S=null;

        System.gc();

        Thread.sleep(5000);

        S.o.println("End of main method");

}

    Public void finalize()

{

        S.o.println("Finalize method Called");

        S=this;

}

%:- 4072869

    finalize method Called

4072869

    End of main method.

Note:- The behaviour of the G.C is vendor dependent & hence we can't exactly because of this we can't answer the following questions exactly.

- ① When JVM runs G.C exactly.
- ② What is the Algorithm followed by G.C.?
- ③ In which order G.C destroys the objects.
- ④ Whether G.C destroys all eligible objects or not etc.

Note:- We can't tell exact algorithm followed by G.C, but most of the cases it is mark & sweep algorithm.

### Memory leak :-

- If an object having the reference then it is not eligible for G.C, even though we are not using that object in our program.
- Still it is not destroyed by the G.C. Such type of object is called "memory leak". (i.e., memory leak is a useless object which isn't eligible for G.C.)
- We can detect memory leaks by making useless objects for G.C explicitly & by invoking G.C programmatically.



These are monitoring tools for memory leak.

## (20) Assertions (1.4 version)

- (1) Introduction
- \* (2) Assert as Key-word & identifier
- (3) Types of assert statements
- (4) Various Runtime flags
- (5) Appropriate & Inappropriate use of assertions
- (6) Assertion Errors.

### Assertions:

- Very Common way of debugging is using S.o.p statements. But the problem with S.o.p's is after fixing the problem Compulsory we should delete these S.o.p's otherwise these S.o.p's will be executed at Runtime and effects performance & disturbs logging.
- To resolve this problem SUN people introduced Assertions Concept in 1.4 version. Hence the main objective of assertions is to perform debugging.
- The main Advantage of assertions over Sof is after fixing the Problem it is not required to delete assert statements because assertions will be disabled automatically at runtime. based on our requirement we can enable & disable assert statements & By default assertions are disabled.
- Assertions Concept is applicable for development & test environment But not for production Environment.

## Assert as a Keyword & identifier :-

246

→ Assert keyword introduced in 1.4 version, Hence from 1.4 version onwards

We can't use assert as identifier. But Before 1.4 we can use assert as identifier

```
Ex:- class Test
{
 public static void main(String[] args)
 {
 int assert = 10;
 System.out.println(assert);
 }
}
```

→ 1) Javac Test.java

C.E:- as of release 1.4, 'assert' is a keyword, and may not be used as an identifier.

Use -Source 1.3 or lower, to use 'assert' as an identifier.

✓ 2) Javac -Source 1.3 Test.java

```
Java Test ←
 ←
 10
```

## Types of Assert Statements :-

→ There are 2 types of Assert Statement

(1) Simple Version

(2) Augmented version

### (1) Simple Version :-

→ assert(b);      b → should be boolean type

→ If b is true, Then our assumption satisfied & rest of the program will be executed normally.

→ If b is false, then our assumption fails The program will be terminated by raising runtime Exception Saying assertionError. So, that we can able to fix the problem.

Ex:- Class Test

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

```
int x = 10;
```

```
///
```

```
assert (x > 10);
```

```
///
s.o.p (x),
```

```
}
```

① Java Test . java ✓

② Java Test ✓

10

\* ③ Java -ea Test ↪

R.E:- Assertion Error.

## (2) Augmented Version :-

21/2

→ we can augment some description by using augmented version to the Assertion Error.

assert(b) : d;  
Should be boolean type

any description, can be any type. but recommended to use String type.

Ex:- class Test

{

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

{

int x=10;

...

assert(x>10) : "Here x value should be >10 but it is not";

...

S.o.pn(x);

}

① Java Test.java ✓

② Java Test ↪

10

③ Java -ea Test ↪

R.E: Assertion Error: Here x value should be >10 but it is not.

Conclusion(1) :-

assert(e<sub>1</sub>) : e<sub>2</sub>;

→ e<sub>2</sub> will be evaluated iff e<sub>1</sub> is false. i.e if e<sub>1</sub> is True, then e<sub>2</sub> won't be evaluated.

Ex:- Class Test

```

 {
 p.s.v.m(String[] args)
 {
 int x=10;
 ==
 assert(x==10); // +x;
 assert(x>10); ++x;
 ==
 System.out(x);
 }
 }

```

✓ javac Test.java ↪

✓ java Test ↪  
10

✓ java -ea Test ↪  
10

javac Test.java

java Test  
10

java -ea Test

R.E: AssertionError@11

Conclusion(2):

assert(e1) : e2 ;

→ As e2 we can take a method call also but void type method calls are not allowed.

Ex:- Class Test

```

 {
 p.s.v.m(String[] args)
 {
 int x=10;
 ==
 assert(x>10); m();
 ==
 System.out(x);
 }
 public static int m()
 {
 return 8888;
 }
 }

```

✓ javac Test.java ↪

✓ java Test ↪  
10

java -ea Test

R.E: AssertionError@8888

→ If m() return type is void, then we will get Compiletime Error  
 Saying "void type not allowed here."

#### 4) Various Runtime flags:-

① -ea :- To enable assertions in Every non-System class

② -enableassertions :- It is Exactly Same as -ea

③ -da :- To disable assertions in Every non-System class

④ -disableassertions :- Same as -da

⑤ -esa :- To enable assertions in every System class.

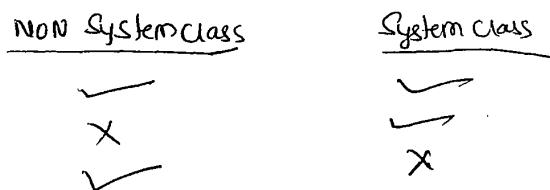
⑥ -enableSystemassertions :- It is exactly Same as -esa.

⑦ -dsa :- To disable assertions in Every System class.

⑧ -disableSystemassertions :- It is Same as -dsa.

Ex(1):-

java -ea -esa -da -dsa -esa -ea -dsa



→ We can use these flags together & all these flags Executed from

Left to Right.

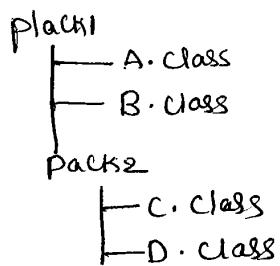
Ex(2):-

① java -ea:pack1.A

② java -ea:pack1.B -ea:pack2.pack3.D

③ java -ea -da:pack1.B

Ex 2:-



→ To enable assertions in only A class

① `java -ea:pack1.A`

→ To enable assertions in Both B & D classes

`Java -ea: pack1.B -ea: pack1.pack2.D`

→ To enable assertions in every non-system class Except B

`Java -ea -da: pack1.B`

→ To enable assertions in every class of pack1 & its Sub packages

`Java -ea: pack1...`

→ To enable assertions in every where in pack1 Except pack2.

`Java -ea: pack1... -da: pack1, pack2...`

### b) Appropriate & Inappropriate use of assertions :-

- i) It is always inappropriate to mix programming logic with assert statement because there is no guarantee of execution of assert statement at runtime.

Ex:-

```
withdraw(int x)
{
 if(x < 100)
 {
 throw new IAG();
 }
}
```

proper way

```
withdraw(int x)
{
 assert(x >= 100);
```

improper way.

2) In our program if there is any place where the control not allowed to reach then it is the best place to use assert statement. 219

Ex:- `switch(x)`

}

`Case1: S.o.println("JAN");`  
`break;`

`Case2: S.o.println("feb");`  
`break;`

⋮

`Case12: S.o.println("Dec");`  
`break;`

`default:`

`} assert(false);`

R-E: A.E can be displayed.

- 3) It is always inappropriate to use assertions for validating public method assignments.
- 4) It is always appropriate to use assertions for validating private method assignments
- 5) It is always inappropriate to use assertions for validating Command-Line assignments because these are assignments to public `main()`.

#### 6) Assertion Error:-

- It is the child class of `Error` & Hence it is unchecked.
- It is legal to catch `AssertionError` by using try-catch but it is stupid kind of activity

Ex:- `class Test`

↓

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

```

Ex:- class Test
{
 p.s.v.m(String[] args)
 {
 int x=10;
 try
 {
 assert(x>10);
 }
 catch(AssertionError e)
 {
 System.out.println("I am Stupid ... b'z I am Catching
 AssertionErro");
 System.out(x);
 }
 }
}

```

### Note:-

→ It is possible to enable assertions either class wise or package wise.

250

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100

14/02/10

25

## Exception Handling

1. Introduction
2. Runtime Stack mechanism.
3. Default Exception Handling.
4. Exception Hierarchy.
5. Customized Exception Handling by Try-Catch.
6. Control flow in Try-Catch.
7. Methods to print exception information.
8. Try with multiple Catch blocks.
9. finally.
10. difference b/w final, finally & finalize.
11. Various possible Combinations of Try-Catch-finally.
12. Control-flow in Try-Catch-finally.
13. Control-flow in Nested Try-Catch-finally.
14. Throws.
15. Throws
16. Exception Handling Keywords Summary.
17. Various possible Compile time Error in Exception handling.
18. Customized Exception.
19. Top-10 Exceptions.

Exception :-

→ when unwanted, unexpected Event that disturbs normal flow of program is called "Exception".

Ex:- Sleeping Exception, Type mismatched Exception, filenotfound - Exception.

→ It is highly recommended to handle Exceptions. The main objective of exception handling is "Gracefull termination of the program".

→ Exception handling does not mean repairing an exception, we have to define alternative way to Continue rest of the program normally, this is nothing but "exception handling".

Ex:- If our programming requirement is to read data from the file located at London & at runtime if that file is not available our program should not be terminated abnormally, we have to provide a local file to Continue rest of the program normally. This is nothing but exception handling.

Syn :- Try

{ read data from London file

}

Catch (FileNotFoundException e)

{

use local file and Continue rest of the program.

normally

}

## Runtime Stack mechanism :-

- For Every Thread JVM will Create a RuntimeStack.
- All the method call performed by the thread will be stored in the Stack.
- Each Entry in the Stack is Called "Activation Record" or Stack frame.
- After Completing Every method Call JVM deletes the Corresponding Entry from the Stack.
- After Completing all method calls, Just before Terminating the Thread JVM destroys the Stack.

Q E.g:-

Class Test

{

  p.s.v.m(String args[])

{

  doStuff();

  {

    p.s.v.dostuff()

}

    doMoreStuff();

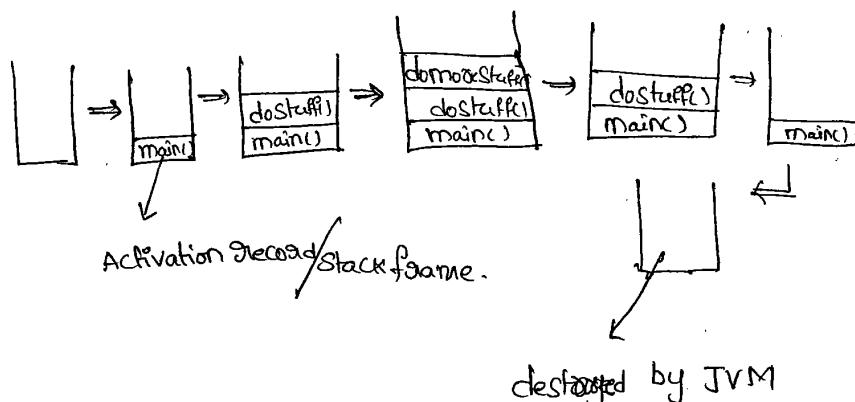
}

    p.s.v.domorestuff()

}

      S.o.println("don't Sleep");

}



## default Exception handling in Java :-

- If any Exception raised, the method in which it is raised is responsible to Create Exception object by including the following information.
  1. Name of Exception
  2. description of Exception.
  3. location of Exception (Stack trace)
- After Creating Exception object, method handovers that Exception object to the JVM.
- JVM checks whether the method Contains any Exception handling Code or not.
- If the method Contains any Exception handling Code, then it will be Executed and Continue rest of the program normally.
- If it doesn't Contain handling code, then JVM terminates that Method abnormally & removes Corresponding Entry from the Stack.
- JVM identifies The Called method & checks whether Called method Contains any handling Code or not. If the Called method doesn't Contain any handling code, then JVM terminates that Called method also abnormally & removes Corresponding Entry from the Stack.
- This process will Continue until main() & If the main() doesn't contain handling Code JVM terminates the main() also abnormally & removes Corresponding Entry from stack.

- 267
- Just before terminating the program abnormally JVM handovers the responsibility of Exception handling to the default Exception handler.
  - Default Exception handler just print exception information to the console in the following format.

|                                 |
|---------------------------------|
| Name of Exception : Description |
| Location (Stack Trace)          |

15/02/11 :-

Class Test

↓

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

↓

doStuff();

↓

P.S.V.doStuff()

↓

doMoreStuff();

↓

P.S.V.~~doMoreStuff()~~

↓

S.O.println(10/0);

↓

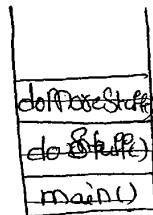
↓

Exception in thread "main" : java.lang.AE : / by zero

at Test.domoreStuff()

at Test.dostuff()

at Test.main()



Runtime Stack

name of exception  
description

stack trace

## Exception hierarchy:-

→ Throwable class acts as a root for entire Java Exception hierarchy.

It has the following 2 child classes

1. Exception

2. Error

### 1. Exception :-

→ Most of the cases Exceptions are caused by our program & these are Recoverable.

### 2. Error :-

→ Most of the cases Errors are not caused by our program these are due to lack of system resources.

→ Errors are NON-Recoverable.

## Checked vs un-checked Exceptions?

→ The exceptions which are checked by compiler for smooth execution of the program at runtime are called 'checked exception.'

Ex:- HallTicketMissing Exception,

PenNotWorking Exception,

FileNotFoundException.

→ The exceptions which are not checked by compiler are called 'un-checked exceptions.'

Ex:- BombBlast Exception.

ArithmaticException, StackExcedentException.

→ Whether Exception is checked or unchecked Composability it should  
runtime only. There is no chance of occurring at Compile time.

→ Runtime Exception and its child classes

→ Errors & its child classes are unchecked Exceptions & all remaining  
are Checked Exceptions

Partially checked vs fully checked :-

→ A checked Exception is Said to be fully checked iff all its child classes  
also checked.

Ex:- IOException

→ A checked Exception is Said to be partially checked iff Some of its  
child classes are unchecked.

Ex:- Exception.

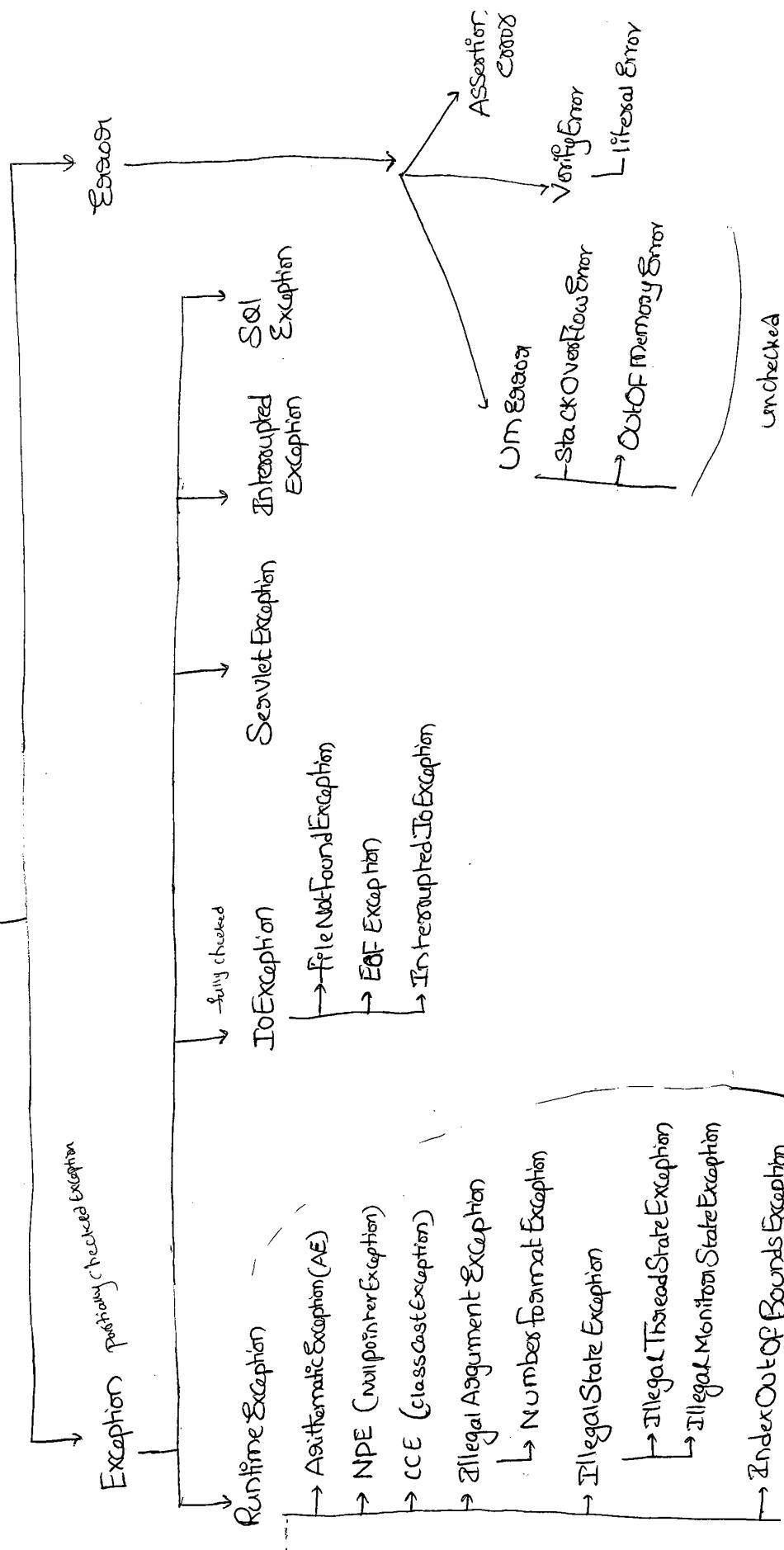
Q) Which of the following are checked

- 1) IOException : fully checked
- 2) Error : unchecked
- 3) Throwable : partially checked
- 4) NullPointerException : unchecked
- 5) InterruptedIOException : fully checked
- 6) ServletException : fully checked.

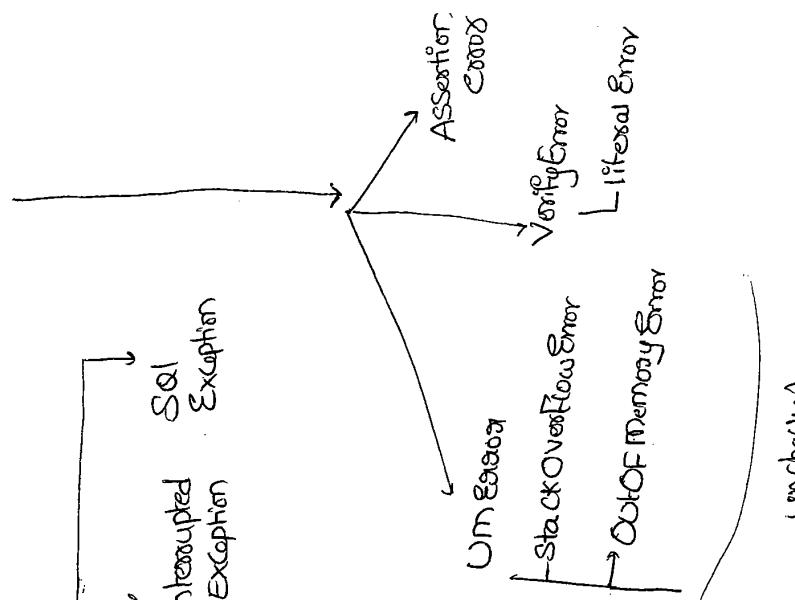
Note:-

→ In Java the only partially checked Exceptions are 1. Exception  
2. Throwable.

## Throwable



## Exception



## Customized Exception Handling by Try-Catch:-

→ We Can maintain Risky Code with in the Try block & Corresponding Handling Code inside Catch block

```
try
{
 RiskyCode;
}
Catch (xxx e)
{
 handling code.
}
```

```
class Test
{
 p.s.v.m (String [] args)
 {
 S.o.pln ("State1");
 S.o.pln (10/0);
 S.o.pln ("State3");
 }
 R.E ≠ A.E : 1 by Zero
 Abnormal termination
}
```

```
class Test
{
 p.s.v.m (String [] args)
 {
 S.o.pln ("State1");
 try
 {
 S.o.pln (10/0);
 }
 Catch (AE e)
 {
 S.o.pln (10/2);
 }
 S.o.pln ("State3");
 }
 O/P : State1
 5
 State3
 Normal termination
}
```

## Control flow in Try-Catch :-

```
try
{
 Stat1;
 State2;
 State3;
}
Catch(*** e)
{
 State4;
}
State5;
```

### Case 1:-

→ If There is no Exception 1, 2, 3, 5 statements are Normal terminations

### Case 2:-

→ If the exception raised at Statement 2 & Corresponding Catchblock matched, 1, 4, 5 are Normal terminations

### Case 3:-

→ If an Exception raised at Statement 2 & The Corresponding Catchblock not matched, 1 followed by Abnormal Termination.

### Case 4:-

→ If an Exception raised at statement 4 or Statement 5 it is always A-N-T

Abnormal Termination

### Note:-

→ Within the Try block if anywhere an Exception raised then rest of the try block won't be Executed even though we handled that Exception. Hence, it is recommended to take only

Risky Code within the Try block. & Length of the Try block should be as less as possible.

2. If an Exception is raised at any Statement which is not part of Try  
Then it is always Abnormal termination.

25b

### Various Methods to print Exception Information :-

16/02/11

→ Throwable class defines the following methods to print Exception information.

(1) printStackTrace():

→ This method prints Exception information in the following format.

Name of Exception : description followed by  
Stack trace

(2) toString():

→ It prints Exception information in the following format.

Name of Exception : description

(3) getMessage():

→ This method prints only description of the Exception.

description

Ego

Class Test

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

1

tag  
J

8.0pt\hbox{10%};

Catch(A-E e)

1

c.pointStackTriangle();

`S.o.p(e); (or) S.o.println(e.toString());`,  $\longrightarrow$  A.E: / by zero

```
s.o.println(e.getMessage());
```

$\rightarrow$  by zero.

Note:-

→ default Exceptionhandler internally uses `printStackTrace()`.

Try with Multiple Catch blocks :-

→ The way of handling an Exception is varied from Exception to Exception  
hence for every Exception it is recommended to take, Separate

Catch block.

Ex!- try

—  
—  
—

J  
Cat C

9  
4

1

( but not recommended . )

```

Ex(8): try
{
 ==
 ==
}
Catch(ArithException e)
{
 Perform these Arithmetic operations;
}

Catch(FileNotFoundException e)
{
 use local file;
}

Catch(NullPointerException e)
{
 use Another resource
}

Catch(Exception e)
{
 default Exception handler;
}

```

✓  
highly recommended

- Hence Try with multiple Catch blocks is possible & highly recommended to use.
- If Try with multiple Catch blocks present then Order of Catch blocks is very important and it should be from child to parent.
- If we are taking from parent to child then we will get Compile time Error saying, "Exception xxxx has already been caught"

Child-to-parent is follows

~  
    ~~try~~  
    {  
        ~~=~~  
        ~~=~~  
        ~~f~~  
    }  
    Catch(exception e)  
    {  
        ~~=~~  
        ~~f~~  
    }  
X  
    Catch(A·E e)  
    {  
        ~~=~~  
        ~~f~~  
    }

```
try
{
 =
 =
}
Catch(A.E e) ✓

Catch(exception e)
{
```

C.E:- Exception java.lang.A.E has already been Caught

finally Block :-

- By

  - It is never recommended to define Clean-up Code with in the block, because there is no guarantee for the Execution of Every Statement.
  - It is never recommended to define Clean-up Code with in the Catch-block, because it won't be Executed if there is no Exception.
  - We required a place to maintain Clean-up code which should be Executed always irrespective of whether exception raised or not raised & whether handle or not handle. Such type of place is nothing but finally-block.
  - Hence, the main purpose of finally-block is to maintain Clean-up Code which should be Executed always.

```

 try
 {
 // Risky Code:
 ...
 catch(XXX e)
 {
 // handling Code;
 ...
 }
 finally
 {
 // Clean-up Code;
 ...
 }
 }

```

Ex@:-

```

class Test
{
 public static void main(String[] args)
 {
 try
 {
 System.out.println("try");
 }
 catch(AE e)
 {
 System.out.println("catch");
 }
 finally
 {
 System.out.println("finally");
 }
 }
}

```

O/P:- try

finally

```

class Test
{
 public static void main(String[] args)
 {
 try
 {
 System.out.println("try");
 System.out.println("10/0");
 }
 catch(AE e)
 {
 System.out.println("catch");
 }
 finally
 {
 System.out.println("finally");
 }
 }
}

```

O/P:- Try  
catch  
finally

```

class Test
{
 public static void main(String[] args)
 {
 try
 {
 System.out.println("try");
 System.out.println("10/0");
 }
 catch(NullPointerException e)
 {
 System.out.println("catch");
 }
 finally
 {
 System.out.println("finally");
 }
 }
}

```

O/P:- try  
finally  
Abnormal

## Return vs Finally:

→ Finally block dominates Return Statement also. Hence, if there is any return statement present inside Try or Catch block, first finally will be Executed & then return Statement will be Considered.

Eg:- Class Test

```
{
 p.s.v.m(Staring [3 args])
}
```

```
try
```

```
{
 S.o.println("try");
}
```

```
return;
```

```
}
```

```
Catch(A.E c)
```

```
{
 S.o.println("catch");
}
```

```
}
```

```
finally
```

```
{
 S.o.println("finally");
}
```

```
}
```

O/P:- try

finally

→ There is only one situation where the finally-block won't be Executed is, when ever JVM shutdown. i.e. when ever we are using System.exit(0).

(\*) Ex:- class Test

259

```
{
 p.s.v.m(String args)
 {
 try
 {
 System.out.println("try");
 System.exit(0);
 }
 catch(AE e)
 {
 System.out.println("catch");
 }
 }
}
```

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

O/P:- tony

\* Difference b/w final, finally & finalize :-

final :-

- It is a modifier applicable for classes, methods & variables.
- If a class declared as final, then child class creation is not possible.
- If a method declared as final, then overriding of that method is not possible.
- If a variable declared as the final, then reassignment is not allowed because, it is a Constant. (changing the value)

finally :-

→ It is block always associated with try-catch to maintain clean-up code which should be executed always irrespective of whether exception raised or not raised & whether handled or not handled.

Finalizers :-

→ It is a method which should be executed by Garbage Collector before destroying any object to perform clean-up activities.

Note:-

→ When Compose with finalize(), it is highly recommended to use finally block to maintain clean-up code. Because, we can't expect exact behaviour of the Garbage Collector.

Various Possible Combinations of try-Catch-finally :-

|                                                                             |                                                                                                          |                                                                         |                                                                                 |                                                                                          |
|-----------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------|---------------------------------------------------------------------------------|------------------------------------------------------------------------------------------|
| ① try<br>{<br>}<br>Catch(ex e)<br>{<br>}<br>}                               | ② try<br>{<br>}<br>Catch(ex e)<br>{<br>}<br>child<br>{<br>}<br>Catch(ex e)<br>{<br>}<br>parent<br>{<br>} | ③ try<br>{<br>}<br>finally<br>{<br>}<br>{<br>}                          | ④ try<br>{<br>}<br>C.E.<br>{<br>}<br>try with out Catch<br>or finally<br>{<br>} | ⑤ X<br>Catch(ex e)<br>{<br>}<br>{<br>}<br>C.E.<br>{<br>}<br>Catch with out try<br>{<br>} |
| ⑥ finally<br>{<br>}<br>X<br>C.E.<br>{<br>}<br>Finally without try<br>{<br>} | ⑦ try<br>{<br>}<br>S.o.pln("Hello");<br>Catch(ex e)<br>{<br>}                                            | ⑧ try<br>{<br>}<br>Catch(ex e)<br>{<br>}<br>S.o.pln("Hello");<br>{<br>} | A   Catch(ex e) C.E. - catch with out try.<br>{<br>}                            |                                                                                          |

⑨ try  
 {  
 }  
 Catch(xx e)  
 {  
 }  
 S.out("Hello");  
 X | finally  
 {  
 }  
 }

C-E! - finally without try

⑩ try  
 {  
 }  
 Catch(xx e)  
 {  
 }  
 finally  
 {  
 }  
 X | finally  
 {  
 }

C-E! - finally without try

⑪ try  
 {  
 }  
 Catch(AE e)  
 {  
 }  
 Catch(Exception e)  
 {  
 }  
 ✓

⑫ try  
 {  
 }  
 Catch(exception e)  
 {  
 }  
 Catch(AE e)  
 {  
 }  
 C-E!  
 Exception Java.lang.AE has  
 already been Caught

⑬ try  
 {  
 }  
 Catch(AE e)  
 {  
 }  
 Catch(AE e)  
 {  
 }

C-E!  
 Exception Java.lang.AE has  
 already been Caught

⑭ try  
 {  
 }  
 Catch(xx e)  
 {  
 }  
 try  
 {  
 }  
 Catch(yes e)  
 {  
 }



⑮ try  
 {  
 }  
 Catch(xx e)  
 {  
 }  
 finally  
 {  
 }  
 try  
 {  
 }  
 catch(yes e)  
 {  
 }



⑯ try  
 {  
 }  
 try  
 {  
 }  
 catch(xx e)  
 {  
 }

C-E! - try without catch or  
 finally

⑰ try  
 {  
 }  
 finally  
 {  
 }  
 X | Catch(x e)  
 {  
 }  
 C-E! - catch without try.

260

## Control flow in try-catch-finally :-

```
try
{
 State 1;
 State 2;
 State 3;
}
Catch (xxx e)
{
 State 4;
}
finally
{
 Statement 5;
}
Statement 6;
```

### Case 1 :-

→ If there is no Exception, then 1, 2, 3, 5, 6, normal termination.

### Case 2 :-

→ If an Exception raised at Statement 2 & the Corresponding Catch-block matched. 1, 4, 5, 6, normal termination.

### Case 3 :-

→ If an Exception raised at Statement 2 & The Corresponding Catch-block not matched. 1, 5, Abnormal termination.

### Case 4 :-

→ If an Exception raised at Statement 4, Then it is always abnormal termination but before that finally block to be Executed.

### Case 5 :-

→ If an Exception raised at State 5 or State 6, It is always abnormal termination.

## Control flow in Nested try-catch-finally :-

261

try

↓  
State 1;  
State 2;  
State 3;

try

↓  
State 4;  
State 5;  
State 6;  
}

Catch(xx e)

{  
    State 7;  
    }  
finally

↓  
State 8;

}

State 9;

}

Catch(yy e)

{

    State 10;

{

    finally

{

        State 11;

{

        State 12;

{

### Case 1:-

→ If there is no Exception, Then 1, 2, 3, 4, 5, 6, 8, 9, 11, 12, Normal termination

### Case 2:-

→ If an Exception raised at Statement 2 and Corresponding Catch block matched. Then 1, 10, 11, 12, Normal termination

### Case 3:-

→ If an Exception raised at Statement 2 and Corresponding Catch blocks not matched. Then 1, 11, Abnormal termination.

### Case 4:-

→ If an Exception raised at Statement 5 & Corresponding inner Catch has matched 1, 2, 3, 4, 7, 8, 9, 11, 12, Normal termination.

### Case 5:-

→ If an Exception raised at Statement 5 & Corresponding inner Catch has not matched but outer Catch has matched. Then 1, 2, 3, 4, 8, 10, 11, 12, Normal

### Case 6:-

→ If an Exception raised at State 5 & inner & outer Catch blocks are not matched Then 1, 2, 3, 4, 8, 11, Abnormal

### Case 7:-

→ If an Exception raised at State 7 & Corresponding Catch blocks matched Then 1, 2, 3, ..., 8, 10, 11, 12, Normal

### Case 8:-

→ If an Exception raised at Statement 7 & The Corresponding Catch not matched Then 1, 2, 3, ..., 8, 11, Abnormal

### Case 9:-

26<sup>v</sup>

→ If an Exception raised at State 8 & Corresponding Catch matched

Then 1, 2, 3 ..., 10, 11, 12, Normal

### Case 10:-

→ If an Exception raised at State 8 & Corresponding Catch has not matched

Then 1, 2, 3 ..., 11, Abnormal

### Case 11:-

→ If an Exception raised at State 9 & Corresponding Catch matched.

Then 1, 2, 3, ..., 8, 10, 11, 12, Normal

### Case 12:-

→ If an Exception raised at State 9 & Corresponding Catch block not matched Then 1, 2, 3, ..., 8, 11, Abnormal

### Case 13:-

→ If an Exception raised at State 10 it is always Abnormal termination  
but before the finally-block will be executed.

### Case 14:-

→ If an Exception raised at State 11 or State 12 it is always Abnormal termination.

10/11

## Throw :-

- Sometimes we can Create Exception Object manually & hand-over that object to the JVM Explicitly by using throw keyword.

throw New ArithmeticException(" / by zero");

To hand-over created exception object to the JVM manually.

↓  
Creation of A.E object Explicitly

- Hence, the main purpose of throw key-word is to hand-over our created exception object manually to the JVM.
- The Result of following two programs is Exactly Same.

```
class Test
{
 p.s.v.m(String [] args)
 {
 S.o.pn (10/0);
 }
}
```

In this Case A.E object created internally & hand-over that object automatically by the main().

```
class Test
{
 p.s.v.m (String [] args)
 {
 throw New ArithmeticException(" / by
 zero");
 }
}
```

In this Case we Created A.E object and we hand-over it to the JVM manually by using throw-keyword.

→ In General, we can use throw keyword for customized Exceptions 263

### Case 1:

→ If we are trying to throw null reference, we will get NullPointerException

```
class Test
{
 static A·E e;
 p·s·v·m (String [] args)
 {
 throw e;
 }
}
```

R.E:- NPE

```
class Test
{
 static A·E e = new A·E ();
 p·s·v·m (String [] args)
 {
 throw e;
 }
}
```

R.E:- A·E

### Case 2:

→ After throw Statement we are not allowed to write any statement

directly otherwise we will get Compiler Compiletime Error Saying

'Unreachable Statement'

```
class Test
{
 p·s·v·m (String [] args)
 {
 S·o·p·ln (10/0);
 }
}
```

S·o·p·ln ("Hello");

}

R.E!:- AE / by zero

```
class Test
{
 p·s·v·m (String [] args)
 {
 throw new A·E (" / by zero ");
 }
}
```

S·o·p·ln ("Hello");

}

C.E!:- unreachable Statement.

### Case 3 :-

→ We can use throw keyword Only for throwable type otherwise we will get Compiletime Error Saying Incompatible Type.

Class Test



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



    throw new Test();



C.E: Incompatible Types

→ Found : Test

Required : Java.lang.Throwable

Class Test extends RuntimeException



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



    throw new Test();



R.E:

Exception in Thread

main : Test

### Throws :-

→ In our program, if there is any chance of raising checked exception Compulsory we should handle it, otherwise we will get compiletime Error Says "unreported Exception xxxx must be caught or declare to be thrown".

Ex:- class Test



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



    Thread.sleep (5000);



C.E:- unreported exception java.lang.IF must be caught

→ we can handle this by using the following two ways.

(1) By Using Try-catch

(2) " " throws

(1) By Using Try-catch:-

Class Test

{

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

{

try

{

Thread.sleep(5000);

}

Catch (I.E e)



{

}

(2) By Using throws Keyword! -

→ we can use throws keyword to delegate the responsibility of exception handling to the ~~handler~~ caller method.

class Test

{

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

{

Thread.sleep(5000);



{

}

- Hence, the main purpose of throws keyword is to delegate responsibility of exception handling to the caller methods in the case of checked exception, to convince compiler.
- In the case of unchecked exceptions, it is not required to use throws keyword.

Eg: Class Test

```
p.s.v.m (String [] args) throws IE
{
 doStuff();
 p.s.v. doStuff() throws IE
 {
 doMoreStuff();
 p.s.v. doMoreStuff() throws IE
 {
 Thread.sleep(5000);
 }
 }
}
```



- In the above program, If we are removing any throws keyword the code won't be compiled. Compulsory we should use 3 throws statements.

18/10/11

265

We can use throws keyword only for Throwable types  
 otherwise we will get Compile time Error saying, incompatible types

```
class Test
{
 p.v.m1() throws test
}

C.E! - incompatible type
 found : Test
 Required : java.lang.Throwable
```

```
class Test extends Throwable Exception
{
 p.v.m1() throws test
}
```

### Case(1) :-

```
class Test (checked)
{
 p.s.v.m(String[] args)
 {
 throw new Exception();
 }
}
```

CE:- undeclared Exception java.lang.Exception must be caught at declared to be thrown.

→ AS Exception is checked Compulsory  
 We should handle either by Try-Catch or by throws keyword

```
class Test (unchecked)
{
 p.s.v.m(String[] args)
 {
 throw new Error();
 }
}
```

R.E!:- Exception in thread "main" java.lang.Error.

→ AS Error is unchecked, it is not required to handle by Try-Catch or by throws

### Case 2!

→ In our program, if there is no chance of raising an Exception  
 Then, ~~it is~~ we can't define Catch block for that Exception.  
 otherwise we will get Compiletime Error, but this rule is applicable  
 for only fully checked Exceptions.

Ex:

```
try
{
 System.out.println("Hello");
}
catch(AE e)
{
}
}Hello
```

```
try
{
 System.out.println("Hello");
}
catch(Exception e)
{
}
}Hello
```

```
try X
{
 System.out.println("Hello");
}
catch(IOException e)
{
}
}Hello
```

```
try X
{
 System.out.println("Hello");
}
catch(InterruptedException e)
{
}
}Hello
```

C.E! - Exception java.lang.Exception is never thrown in body of corresponding try Statement.

```
try ✓
{
 System.out.println("Hello");
}
catch(Exception e)
{
}
}Hello
```

### Keywords for Exception:

- try
- catch
- finally
- throw
- throws

## Exception Handling Keywords Summary :-

- 1) try :- To maintain Risky Code.
- 2) Catch :- To maintain Handling Code.
- 3) Finally :- To maintain Clean-up Code.
- 4) throw :- To hand-over Our Created Exception Object to the JVM manually.
- 5) throws :- To delegate The Responsibility.

## Various Possible Complieetime Errors in Exception Handling:-

- ① Exception xxxx has already been caught (try with multiple catch)
- ② Unreported Exception xxxx must be caught or declared to be thrown
- ③ Exception xxxx is never thrown in body of corresponding try statement
- ④ try without Catch or Finally
- ⑤ finally without try
- ⑥ Catch without try
- ⑦ unreachable Statement
- ⑧ Incomplatable types

found : Test

Required : Java.lang.Throwable.

## Customized Exceptions:

→ To meet our programming requirement sometimes we have to create our own Exceptions. Such types of Exceptions are called "Customized Exceptions".  
Ex: TooYoungException, TooOldException, InsufficientFundsException..etc

Class TooYoungException extends RuntimeException

{

TooYoungException(String s)

{

Super(s);

}

Class TooOldException extends RuntimeException

{

TooOldException(String s)

{

Super(s);

}

class Test

{

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

{

int age = Integer.parseInt(args[0]);

if (age > 60)

{

throw new TooYoungException("plz wait some more time") ||  
age is already crossed marriage age.

}

else if (age < 18)

{

throw new TooYoungException("Ur age is already crossed marriage  
age... no chance of getting married") ;

else  
↓

Sopln("you will get match details by mark");

}

}.

### Note:-

→ It is highly recommended to keep our Customized Exception class as unchecked, i.e. we have to Extend Runtime Exception Class but Not Exception Class while defining our customized Exceptions.

### Top-10 Exceptions :-

21-02-11

- Based on the Source, who triggers the Exception, all Exceptions are divided into 2 types.
- 1. J.V.M Exceptions
- 2. programmatic Exceptions.

#### 1. JVM Exceptions :-

- The Exceptions which are raised automatically by the JVM when even a particular Event occurs are Called JVM Exceptions.
- Ex:- (i) ArrayIndexOutOfBoundsException.
- (ii) NullPointerException.

#### 2. programmatic Exceptions :-

- The Exceptions which are raised Explicitly either by the programmer or by the API developer, are Called programmatic Exception.
- Ex:- IllegalAssignmentException, NumberFormatException.

## (1) ArrayIndexOutOfBoundsException :-

→ It is the child class of RuntimeException & hence it is unchecked.

→ Raised automatically by the JVM, whenever we are trying to access array element with out of range index.

Ex:- `int [] a = new int[10];`

`s.o.println(a[0]);` o ✓

`s.o.println(a[100]);` RE:- AIOOBE

## (2) NullPointerException :-

→ It is the child class of RuntimeException and hence it is unchecked.

→ Raised automatically by the JVM, whenever we are trying to access perform any operation on null.

Ex:- `String s = null;`

`s.o.p(s.length());` RE:- NPE

## (3) StackOverflowError :-

→ It is the child class of Error and hence it is unchecked.

→ Raised automatically by the JVM, whenever we are trying to perform recursive method invocation.

Ex:- Class Test

↓  
P-S-V-m m1()

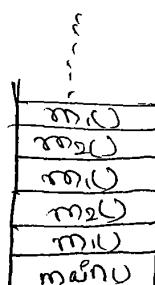
↓  
m2()

↓  
P-S-V-m m2()  
↓ m1(); ↓

P-S-V-m (String c1 args)

↓  
m1();

↓  
m1();



RE:- SOFE.

#### (4) NoClassDefFoundError :-

26/6

- It is the child class of Error and hence it is unchecked.
- Raised automatically by the JVM, whenever JVM unable to find required class.

Ex:- Java Sainu ↪

- If Sainu.class file is not available then we will get R.E Saying NO ClassDefFoundError.

#### (5) ClassCastException :-

- It is the child class of RuntimeException and hence it is unchecked.
- Raised automatically by JVM whenever we are trying to typeCast parent object to the child type.

Ex:-  
✓ String s = new String("duaga");  
Object o = (Object) s;

~~Object~~ Object o = new Object(); | X  
String s = (String) o;

R.E:- CCE

#### (6) ExceptionInInitializerError :-

- It is the child class of Error and hence, it is unchecked.
- Raised automatically by the JVM, if any Exception occurs while performing initialization for static variables and <sup>while</sup> executing static blocks.

Ex:-

Class Test

{

    Static int i = 10/0;

}

R.E:-

ExceptionInInitializationError

Caused by java.lang.AE :/ by zero.

Class Test

{

    Static

}

    String s=null;

    S.length();

}

R.E:- ExceptionInInitializationError

Caused by java.lang.NPE

### ③ Illegal Assignment Exception :-

→ SE is the child class of RE & hence it is unchecked.

→ Raised Explicitly by the programmer or by API developer

to indicate that a method has been invoked with invalid assignment

Ex:-

Thread t = new Thread();

t.setPriority(10); ✓

t.setPriority(100); ✗ R.E:- IAE

### ④ NumberFormatException

→ SE is the child class of RE & hence it is unchecked.

→ Raised Explicitly by the programmer or by API developer

to indicate that we are trying to convert String to number type

but the String is not properly formatted

Ex:- ✓ int i = Integer.parseInt("10");

✗ int i = Integer.parseInt("ten"); R.E:- NFE



### ⑨ IllegalStateException :-

- It is the child class of RuntimeException and hence, it is unchecked.
- Raised Explicitly by the programmer or by the API developer to indicate that a method has been invoked at inappropriate time.

Ex:-

Once Session Expires we Can't Call any method on that object  
Otherwise we will get IllegalStateException.

Ex ①:  

```
HttpSession session = req.getSession();
System.out.println(session.getId()); 123456789
```

~~X~~

```
session.invalidate();
System.out.println(session.getId()); R.E:- ISE
```

Ex ②:-

Thread t = new Thread();

t.start();

~~X~~ t.start(); R.E:- IllegalThreadStateException.

- After Starting a thread, we are not allowed to restart the same thread, otherwise we will get R.E:- IllegalThreadStateException

## 10) AssertionError

- It is the child class of Error & hence it is unchecked.
- Raised Explicitly either by the programmer or by API developer  
to indicate that ~~a method has~~ assert statement fails.

Ex:- `AssertionError`;

R.E:- Assertion Error.

| <u>Exception/Error</u>         | <u>Raised by</u>                        |
|--------------------------------|-----------------------------------------|
| 1. AIOOBE                      |                                         |
| 2. NPE                         |                                         |
| 3. SOFE                        | JVM automatically                       |
| 4. NoClassDefFoundError        | (JVM Exception)                         |
| 5. ClassCastException          |                                         |
| 6. ExceptionInInitializerError |                                         |
| 7. IllegalArgumentExcepton     |                                         |
| 8. NumberFormatExcepton        | either programmer or API developer      |
| 9. IllegalStateException       | Explicitly<br>(Programmatic Exceptions) |

## Exception Propagation-

- The process of delegating the Responsibility Exception handling from one method to another method by using throws keyword is called Exception propagation

## Inner classes

- Sometimes we can declare a class inside another class, Such type of classes are called Innerclasses:
- Innerclasses Concept introduced in Java 1.1 version to fix GUI bugs as the part of Eventhandling.
- But Because of powerful features & benefits of Innerclasses slowly programmers started using even in regular coding also.
- without existing one type of object if there is no chance of existing another type object, then we should go for Inner class concept.

Ex(1):-

→ without existing Car object, if there is no chance of existing wheel object then we should go for Inner classes.

→ we have to declare wheel class with in the Car class.

```
class Car
```

```
{
```

```
 class Wheel
```

```
{
```

```
}
```

```
}
```

②:- without existing Bank object there is no chance of existing Account object, hence we have to define account class inside Bank class.

```
class Bank
```

```
{
```

```
 class Account
```

```
{
```

```
}
```

(3) → A map is a collection of Key Value pairs and each key-value pair is called Entry. Without existing map object there is no chance of existing Entry object. Hence interface Entry is defined inside Map interface.

Interface Map

{

    Interface Entry

{

}  
}

Note:-

→ The relationship b/w Outer & Inner classes is not parent to child

Relationship. It is has-A Relationship.

→ Based on the purpose & position of declarations all inner classes are divided into 4 types

- 1) Normal or Regular Inner classes.
- 2) Method Local Inner classes
- 3) Anonymous Inner classes (without class name)
- 4) Static Nested Classes

<sup>note</sup>  $\Rightarrow$   
Note:-

From Static Nested Class we can access only static members of outer class directly. But in normal Inner classes we can access both static & non-static members of outer class directly.

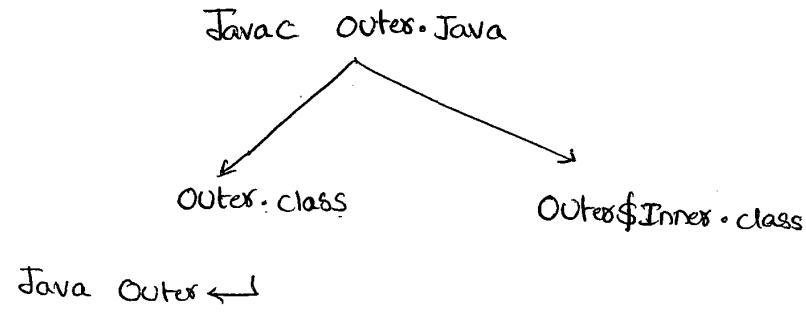
## Normal or Regular Inner class :-

271

→ If we declare any named class directly Inside a class without static modifier, Such type of class is called "Normal or Regular Inner Class"

Ex(1):-

```
Class Outer
{
 Class Inner
 {
 }
}
```



R.E:- NoSuchMethodError : main

Java Outer\$Inner

R.E:- NoSuchMethodError : main

Ex(2):-

```
Class Outer
{
 Class Inner
 {
 }
 public static void main(String[] args)
 {
 System.out.println("Outer class main method");
 }
}
```

%!- Javac Outer.java

Java Outer ←

%!- outer class main method.

Java Outer\$Inner ←

%!- NoSuchMethodError : main.

Ex(3) :-

→ Inside Inner classes we can't declare static members hence it is not possible to declare main method & hence we can't invoke inner class directly from Command prompt.

Ex:- class Outer

{

    class Inner

{

    p.s.v.m(String [] args)

X

    s.o.println("inner class method");

}

}

Javac Outer.java

C.E:- Inner Classes Can't have static declarations

Result:-

Accessing Inner class Code from Static area of outer Class:-

Ex:- class Outer

{

    class Inner

{

    p.s.v.m()

{

    s.o.println("Inner class method");

}

}

    p.s.v.m(String [] args)

{

        Outer o = new Outer();

        Outer.Inner i = o.new Inner();

```
i. m1();
}
}
```

Q1. Java Outer.java ↗

java Outer ←

Inner class method.

```
Outer o = new Outer();
Outer.Inner i = o.new Inner();
i.m1();
```

Accessing Inner class Code from Instance Area of Outer Class:-

Ex. Class Outer

```
class Outer
{
 class Inner
 {
 p.v.m1()
 {
 System.out.println("Inner class method");
 }
 }
}
```

```
p.v.m2()
```

```
Inner i = new Inner();
```

```
i.m1();
```

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

```
Outer o = new Outer();
```

```
o.m2();
```

## Accessing Inner class Code from Outside of outer class

Ex:

Class Outer

{

Class Inner

{

p.v.m1();

{

Saying ("Inner class method");

{

}

}

Class Test

{

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

{

Outer o = new Outer();

)

Outer.Inner i = o.new

)

Inner();

)

i.m1();

{

}

### Accessing Inner Class Code

from static area of Outer class

(a)

from outside of outer class

Outer o = new Outer();

Outer.Inner i = o.new Inner();

i.m1();

from instance area of  
outer class

Inner i = new Inner();

i.m1();

Outer o = new Outer();

→ From the Inner class we can access all members of outer class  
(both static & non-static) directly.

973

Ex:- Class Outer

```
 {
 static int x=10;
 int y = 20;
```

Class Inner

```
 {
 public void m1()
 {
 System.out.println(x); 10
 System.out.println(y); 20
 }
 }
```



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

```
↓
new Outer().new Inner().m1();
```

```
{
 }
 10
 20
```

With in the Inner class this always pointing to Current Inner class Object.

→ To refer Current Outer class object we have to use "Outerclassname.this".

"Outerclassname.this".

Ex:-

Eg:-

Class Outer

{

int x=10;

Class Inner

{

int x=100;

public void m1()

{

int x=1000;

S.o.println(x); 1000

S.o.println(this.x); 100

S.o.println(outer.this.x); 10

}

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

{

New Outer().NewInner().m1();

}

→ For the Outer classes (Top-level classes) the applicable modifiers are public, <default>, final, abstract, Strictfp.

But for the Inner classes & in addition to above the following modifiers are also applicable.

only for Outer classes

public  
default  
final  
abstract  
Strictfp

+

private  
protected  
static

= Inner classes

## 2) Method Local Inner classes :-

27

- Sometimes we can declare a class inside a method such type of classes are called "Method Local Inner classes".
- The main purpose of method local inner class is to define method specific functionality.
- The scope of method local inner class is the method in which we declared it. That is from outside of the method we can't access method local inner classes.
- As the scope is very less, this type of inner classes are most rarely used inner classes.

Ex:-

```
Class Test
{
 public void m1()
 {
 class Inner
 {
 public void sum(int x, int y)
 {
 System.out.println("Sum is :" + (x+y));
 }
 }

 Inner i = new Inner();
 i.sum(10, 20);
 i.sum(100, 200);
 i.sum(1000, 2000);
 i.sum(10000, 20000);
 }
}
```

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

}

New Test().m1();

}

}

Q1:- Sum is 30

Sum is 300

Sum is 3000

Sum is 30000

→ We can declare Inner class either in Instance method or in Static method.

→ If we declare Inner class inside Instance method then we can access Both Static & Non-Static variables of outer class directly from that Inner class.

→ If we declare Inner class inside Static method then we can access Only Static members of Outer Class directly from that Inner class.

Ex:- class Test

{

int x=10;

Static int y=20;

public void m1()

if Static is there

{

class Inner

{ p. void m2()

S. o. println(x); 10

S. o. println(y); 20

Inner i = new Inner();

i. m2();

O/P :- 10  
20

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

275

```
{
 new Test().m1();
}

O/P:- 10, 20
```

\* From method Local InnerClass we can't access Local variables of the method in which we declared it. But if that local variable declared as the final Then we can access.

Eg:-

```
class Test
{
 int x=10;

 public void m1()
 {
 int y=20; → if we declare final
 { (final int y=20;)
 class Inner
 {
 public void m2()
 {
 System.out.println(x);
 System.out.println(y);
 }
 }
 }
 }
}
```

O/P:- x=10  
y=20

```
Inner i=new Inner();
```

```
i.m2();
```

O/P:-

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

```
{
 new Test().m1();
}
```

C.E:- Local variable y is accessed from with inner class, needs to be declared final.

→ If we declare y as final Then we won't get any Completetime Error.

%pl-  
x= 10  
y= 20

24/02/11:

Q) Consider the following Code

```
class Test
{
 int x=10;
 static int y=20;
 public void m1()
 {
 int i=30;
 final int j=40;
```

```
class Inner
{
 public void m2()
```

—————> Line①  
} } }

→ At line① which Variables we can access      ① x ✓  
                                                        ② y ✓  
                                                        ③ i ✗  
                                                        ④ j ✓

Note!:- If declare m1() as static Then at Line① which variables we can access as y, j .

⑤ If we declare  $m_2()$  as static, then which variable <sup>we can</sup> access Line ①  
we will get C.E. because Inside Inner classes we can't have static declarations.

→ The only applicable modifiers for method Local Inner classes are final, abstract, strictfp,

### ③ Anonymous Inner Class :-

→ Sometimes we can declare a class without name also. Such type of nameless inner classes are called Anonymous Inner classes.

→ This type of inner classes are most commonly used type of inner classes.

→ There are 3 types of Anonymous inner classes.

1. Anonymous inner class that extends a class.

2. " " " implements an interface.

3. " " " defined inside method assignments.

### ④ Anonymous inner class that extends a class :-

Ex:- Class Popcorn

    public void taste()

        System.out.println("Salty");

    }

    // 100 more methods

Class Test

    P.S.V.M (String [ ] ans)

    ↓

Popcorn p = new Popcorn()

{

    public void taste()

    ↓

    System.out.println("Sweet");

    }

p.taste(); Sweet

Popcorn p<sub>1</sub> = new Popcorn();

p<sub>1</sub>.taste(); Salty.

↓

### Note:-

- ① The internal class name generated for Anonymous Inner class is "Test\$1.class".
- ② Parent class Reference can be used to hold child class object but by using that reference we can call only methods available in the Parent class & we can't call child specific methods. In the anonymous inner classes also we can define new methods but we can't call these method from outside of the class because ~~these are~~ we are depending on parent reference. This methods just for internal purpose only.

### Analysis:-

```
popcorn p = new Popcorn();
```

→ Just we are creating an object of Popcorn class.

→ Popcorn p = new Popcorn()  
    {  
        };

→ We are creating child class for the Popcorn & for that child class we are creating an object with parent reference.

Q1.

2x8

```
class Test
{
 p.s.v.m (String [] args)
 {
 Thread t = new Thread()
 {
 p.v.run()
 {
 for (int i=0 ; i<10 ; i++)
 {
 System.out.println("child thread");
 }
 }
 t.start();
 for (int i=0 ; i<10 ; i++)
 {
 System.out.println("main thread");
 }
 };
 }
}
```

→ In the above Example both main & child threads will be Executed Simultaneously & Hence we can't ~~get~~ exact output.

(b) Anonymous Inner Class That Implements an Interface!

Ex:-

Class Test

{  
    p.s.v.m (String [] args)

}  
Runnable a = new Runnable()

{  
    public void run()

{  
    for (int i=0 ; i<10 ; i++)

{  
    System.out.println("child Thread");

}  
};

Thread t = new Thread(a);

t.start();

for (int i=0 ; i<10 ; i++)

{  
    System.out.println("main Thread");

}.

→ It is an object of  
Runnable

(c) Anonymous Inner Class that define & Inside method assignment:-

Eg:-

Class Test

↓

Public static void main (String [] args)

↓

New Thread (new Runnable()

↓

public void run()

↓

for (int i=0 ; i<10 ; i++)

↓

System.out.println ("child thread-i");

↓  
}).start();

for (int i=0 ; i<10 ; i++)

↓

System.out.println ("main thread-i");

↓  
}

## General class Vs Anonymous Inner class:-

28/02/11

- A General class Can extend only one class at a time. whereas as Anonymous Innerclass also Can extend only one class at a time.
- A General class Can implement any no. of Interfaces whereas Anonymous Innerclass Can implement only one interface at a time.
- A General class Can Extend another class & Can Implement an interface Simultaneously. whereas as Anonymous Inner class Can extend another or Can implement an interface but not both simultaneously.

## Static Nested classes:-

- Some times we can declare Inner class with static modifier. Such type of Inner classes are called "Static Nested classes".
- In the normal Inner class, Inner class object always associated with outer class object.
- i.e., without existing outer class object, there is no chance of existing Inner class object.
- But static Nested class object is not associated with Outer class object i.e. without existing outer class object there may be a chance of existing static Nested class object.

Eg:- class Outer

{

    Static class Nested

{

    Public void m1()

{

        System.out.println("Static Nested class method");

}

28

```
public static void main(String[] args)
{
 Outer.Nested n = new Outer.Nested();
 n.m1();
}
```

→ Within the Static Nested Class we can declare static members including main() also. Hence it is possible to invoke Nested class directly from Command prompt.

Ex:

```
Class Outer
{
 static class Nested
 {
 public static void main(String[] args)
 {
 System.out.println("Static Nested class main method");
 }
 }

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

javac Outer.java  
java Outer  
Outer class main method  
Java Outer\$Nested  
Static Nested Class main method

→ From the Normal Inner class both Static & Non-static members directly.

but from, Static Nested class we can access only static members of outer class directly.

Ex:- class Outer

{

int x=10;

static int y=20;

static class Nested

X

{

p.v.m1();

{

S.o.println(x); \* → C.E :- Non-Static variable x can't be

S.o.println(y); ↘

referenced from Static Nested Content

{

}

Diff b/w Normal Innerclass  
Innerclass

& Static Nested Class?

Static Nested Class

- 1) Inner class object is always associated with Outer class object.  
i.e without existing Outer class object there is no chance of existing Inner class object

2) Inside Normal Inner class we can't declare static members

3) Inside normal Inner class we can't declare main() and hence it is not possible to invoke inner class directly from Command prompt

1) Static Nested Class object is not associated with Outer class object,  
i.e, without existing Outer class object there may be a chance of existing Static Nested class object:

2) Inside Static Nested class we can declare static members.

3) Inside Static Nested class we can declare main() & hence we can invoke Static Nested class directly from Command prompt

## Java.lang package

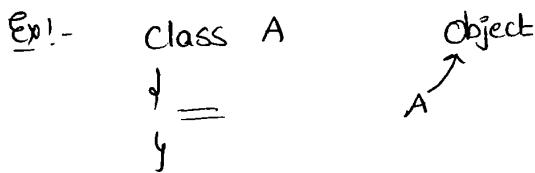
281

- The most commonly required classes & Interfaces which are required for writing any java program whether it is simple or complex, are encapsulated into a separate package which is nothing but lang package
- It is not required to import lang package explicitly because by default it is available to every java program.
- The following are some of the commonly used classes in lang package

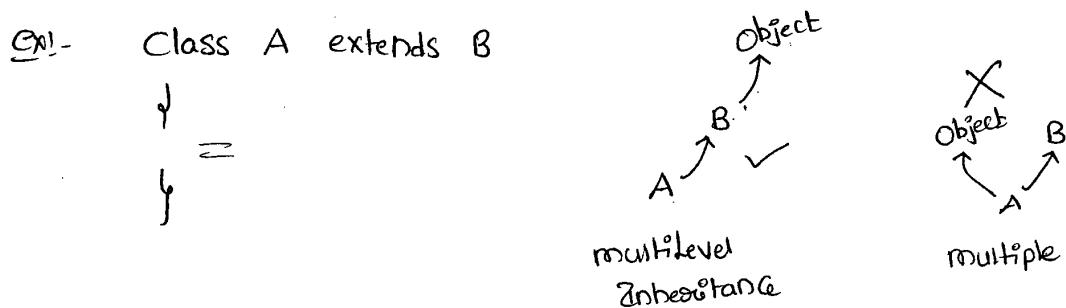
- ① Object
- ② String
- ③ StringBuilder
- ④ StringBuffer
- ⑤ Autoboxed classes (Auto boxing & Auto unboxing)

### ① Object :-

- The most common methods which are required for any java object are encapsulated into a separate class which is nothing but object class.
- SUN people made this class as parent for all Java classes so that its methods are by default available to every Java class automatically.
- Every class in java is the child class of object either directly or indirectly, if our class don't extend any other class then only our class is direct child class of object.



→ If our class extends any other class then our class is not direct child class of Object. It extends Object class indirectly.



→ Object class defines the following 11 methods

- (1) public static toString();
- (2) public native int hashCode();
- (3) public boolean equals(Object o);
- (4) protected native Object clone() throws CloneNotSupportedException;
- (5) public final Class getClass();
- (6) protected void finalize() throws Throwable;
- (7) public final void wait() throws InterruptedException;
- (8) public final native void wait(long ms) throws IE;
- (9) public final native void wait(long ms, int ns) throws IE;
- (10) public final native void notify();
- (11) public final native void notifyAll();

## ① toString() method :-

Q8 ✓

- We can use this method to find String representation of an object
- Whenever we are trying to print any object reference internally `toString()` method will be executed.

Ex:-

Class Student

{

String name;

int rollno;

Student (String name, int rollno)

{

this.name = name;

this.rollno = rollno;

{

P: S. v. m (String args)

{

Student s<sub>1</sub> = new Student ("durga", 101); ✓

Student s<sub>2</sub> = new Student ("Santu", 102); ✓

S.o.println(s<sub>1</sub>);  $\xrightarrow{\text{to}}$  S.o.println(s<sub>1</sub>.toString()); Student @ 3e25a5

S.o.println(s<sub>2</sub>); Student @ 19821f.

{

{

- In the above Case Object class `toString()` method got executed which is implemented as follows.

```

public String toString()
{
 return getClass().getName() + "@" + Integer.toHexString(hashCode());
}

```

Student @ 3e25a5

- ~~To~~
- Classname@hexadecimal String representation of hash code.
- To provide our own String representation we have to override toString() in our class which is highly recommended.
- whenever we are trying to print Student Object reference to return his name & roll number we have to override toString() as follows

```

public String toString()
{
 // return Name;
 // return Name + "----" + roll no;
 // return "this is Student with name:" + name + ", with rollno:" + roll no;
}

```

- \* In String, StringBuffer & all wrapper classes toString() method is overridden to return proper String form. Hence, it is highly recommended to override toString() method in our class also.

Ex :- Class Test

```

 |
 |
 |public String toString()
 |
 | return "Test";
 |
 |}
 |
 |public . s. v. m(→)
 |
 |}
 |
 |Test t = new Test();

```

String s = new String("durga");

Integer i = new Integer(10);

s.o.println(t); test

test @a235a4

s.o.println(s); durga ✓

s.o.println(i); 10

} }

### (ii) hashCode() :

→ For every object Jvm ~~will always~~ will assign one unique id.

which is nothing but hashCode.

→ Jvm uses hashCode, will saving objects into hashtable or hashSet or hashmap

→ Based on our requirement we can generate hashCode by overriding hashCode method in our class.

→ If we are not overriding hashCode() method then Object class

`hashCode()` method will be executed which generates `hashCode` based on Address of the Object But whenever we are overriding `hashCode()` method then `hashCode` is no longer related to Address of the Object.

→ Overriding `hashCode()` method is said to be proper iff for every object we have to generate a unique number.

Ex:- Case ①:-

Class Student

↓

==  
==

public int hashCode()  
↓  
return 100;

↓

! :

Case ②:-

Class Student

↓

==  
==

public int hashCode()  
↓  
return \* student;

↓

! :

Case ③:- It is improper way of overriding `hashCode()` because we are generating same `hashCode` for every object

Case ④:-

It is proper way of overriding `hashCode()` because we are generating a different `hashCode` for every object

## toString() Vs hashCode() :-

28/1

Ex:-

Class Test

↓

int i;

Test(int i)

↓

this.i = i;

↳

p.s.v.m(—)

↳

Test t<sub>1</sub> = new Test(10);

↳

Test t<sub>2</sub> = new Test(100);

↳

S.o.println(t<sub>1</sub>); Test@1a3b2b

↳

S.o.println(t<sub>2</sub>); Test@2a4b2a

↳

Object → toString()

↓

Object → hashCode()

0-15

0

1

2

3

4

a(10)

b(11)

c(12)

d(13)

e(14)

f(15)

$$\begin{array}{r} 100 \\ \hline 16 \end{array}$$

6 - 4

64

10

$$\begin{array}{r} 100 \\ \hline 16 \end{array}$$

64 - 47

$$\begin{array}{r} 53 \\ \hline 16 \end{array}$$

64

10 → a

in hashCode

Object → toString()

↓

Test → hashCode()

10 → a

in hashCode

Ex Q1) Class Test

↓

int i;

Test(int i)

↓

this.i = i;

↳

public int hashCode()

↓

return i;

↳

p.s.v.m(—)

↓

Test t<sub>1</sub> = new Test(10);

Test t<sub>2</sub> = new Test(100);

S.o.println(t<sub>1</sub>); Test@ a

S.o.println(t<sub>2</sub>); Test@ b

↳

Object → toString()

↓

Test → hashCode()

10 → a

in hashCode

ex3:-

Class Test

{

int i;

Test (int i)

{

this.i = i;

}

Public int hashCode()

{

return i;

}

Public String toString()

{

return i + " ";

}

P. S. v. m(\_\_\_\_)

{

Test t<sub>1</sub> = new Test(10);

Test t<sub>2</sub> = new Test(100);

S.o.println(t<sub>1</sub>);      10

S.o.println(t<sub>2</sub>);      100

}

Test → toString()

Note:-

- If we are giving opportunity to Object class → toString() method  
 Then it will call internally hashCode() method.
- If we are giving opportunity to our class toString() method  
 Then it may not call hashCode() method.

26

### ③ equals() method :-

- We can use equals() method to check equality of two objects

```
public boolean equals(Object o)
```

Ex:- Class Student

↓  
String name;

int rollno;

Student (String name, int rollno)

↓  
this.name = name;

this.rollno = rollno;

{  
P. S. v. m(\_\_\_\_\_)

↓

Student  $s_1$  = new Student ("durga", 101);

Student  $s_2$  = new Student ("pavan", 102);

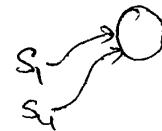
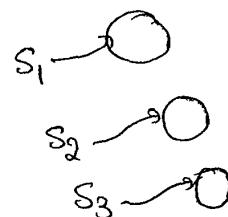
Student  $s_3$  = new Student ("durga", 101);

Student  $s_4$  =  $s_1$ ;

$s_1$ .equals( $s_2$ )); false

$s_1$ .equals( $s_3$ )); false

$s_1$ .equals( $s_4$ )); true



- In the above Case Object class .equals() method will be executed which is always meant for Reference Comparison (address Comparison).
- i.e., if two references pointing to the same object then only .equals() method returns true. This behaviour is exactly same as == operator.
- If we want to perform Content Comparison instead of Reference Comparison we have to override .equals() method in our class.
- Whenever we are overriding .equals() method we have to consider the following things,
  - (1) What is the meaning of equality
  - (2) In the case of diff. Type of Objects (Heterogeneous) equals method should return false but not ClassCastException.
  - (3) If we are passing Null assignment our .equals method should returns false but not a NullPointerException.
- The following is the valid way of overriding equals() method in Student class.

```
exp: public boolean equals(Object o)
 ↓
 try
 ↓
 String name1 = this.name;
 int rollno1 = this.rollno;
 Student s2 = (Student) o;
 Student name2 = s2.name;
 int rollno2 = s2.rollno;
```

ggb

if (name1.equals(name2) && rollno1 == rollno2)

}

    return true;

}

else

}

    return false;

}

Catch (CCE e)

}

    return false;

}

Catch (NPE e)

}

    return false;

}

Student s<sub>1</sub> = new Student ("durga", 101);

Student s<sub>2</sub> = new Student ("paran", 102);

Student s<sub>3</sub> = new Student ("durga", 101);

Student s<sub>4</sub> = s<sub>1</sub>;

s.o.println(s<sub>1</sub>.equals(s<sub>2</sub>));      false

s.o.println(s<sub>1</sub>.equals(s<sub>3</sub>));      true

s.o.println(s<sub>1</sub>.equals(s<sub>4</sub>));      true

s.o.println(s<sub>1</sub>.equals("durga"));      false

s.o.println(s<sub>1</sub>.equals(null));      false

Short way of writing equals() method :-

```
public boolean equals(Object o)
{
 try
 {
 Student s2 = (Student)o;
 if (name.equals(s2.name) && rollno == s2.rollno)
 return true;
 else
 return false;
 }
 catch (ClassCastException e)
 {
 return false;
 }
 catch (NullPointerException e)
 {
 return false;
 }
}
```

Relationship b/w == operator & .equals() method :-

- \* If  $s_1 == s_2$  is True, Then  $s_1.equals(s_2)$  is always True.
- \* If  $s_1 == s_2$  is False, Then we can't expect about  $s_1.equals(s_2)$  Exactly. It may returns True or False.
- \* If  $s_1.equals(s_2)$  returns True, we can't conclude anything about  $s_1 == s_2$ . It may returns either True or False.
- \* If  $s_1.equals(s_2)$  is False, Then  $s_1 == s_2$  is always False.

## differences b/w == operator & .equals() method :-

gft

### == operator

- ① IT IS an operator applicable for both primitives & Object references.
- ② In the Case of Object references,  
== operator is always meant for Reference Comparison. i.e., if two references pointing to the same object.  
Then only == operator returns True.
- ③ we can't override == operator for Content Comparison.
- ④ In the Case of Heterogeneous type Objects == operator gets Causes Compiletime Error  
Saying incompatible types.
- ⑤ for any object reference  $g_1$ ,  
 $g_1 == \text{null}$  is always False.

### .equals()

- ① IT IS a method applicable only for object references but not for primitives.
- ② By default .equals() method present in Object class is also meant for reference Comparison only.
- ③ we can override .equals() method for Content Comparison.
- ④ In the Case of Heterogeneous Objects .equals() method Simply return false & we won't get any Compiletime or Runtime Error.
- ⑤ for any object reference  $g_1$ ,  
 $g_1.equals(\text{null})$  is always False.

Note:-

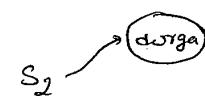
- Q) what is the difference b/w Double Equal operator ( $= =$ ) & `equals()`  
→ " $= =$ " Operator is always meant for Reference Comparison, where  
as `equals()` method meant for Content Comparison.

Ex:-

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



`String s2 = new String ("durga");`



`System.out.println(s1 == s2);`; false

`System.out.println(s1.equals(s2));`; true

→ In `String`, ~~All wrapper classes~~ `equals()` is overridden for Content Comparison.

→ In `StringBuffer` class `equals()` is not overridden for Content Comparison hence object class `equals()` got executed which is meant for reference Comparison.

→ In wrapper class `equals()` is overridden for Content Comparison

Contract b/w `equals()` & `hashCode()` :-

1. If two objects are equal by `equals()` Compulsory their `hashCodes` must be same.

2. If two objects are not equal by `equals()` then there are no restrictions on `hashCode()`, they can be same or different.

3. If `hashCodes` of 2 objects are equal, then we can't conclude above `equals()`, it may returns True or False.

↳ If hashCodes of 2 objects are not equals then we can always conclude .equals() returns false.

### Conclusion :-

→ To Satisfy The above Contract b/w .equals() and hashCode(), whenever we are overriding .equals() Compulsory we should Override hashCode().

→ If we are not overriding we won't get any Compile time & Run-time errors.

→ But it is not a good program practice.

Q) Consider The following .equals()

```
public boolean equals(Object obj)
{
 if(!(obj instanceof person))
 return false;
 person p = (person) obj;
 if (name.equals(p.name) & (age == p.age))
 return true;
 else
 return false;
}
```

Q) Which of the following hashCode() Are Said to be properly implemented.

X ① public int hashCode()
{
 return 100;
}

X ④ public int hashCode()

↓  
return age + (int)height;

y

✓ ③ public int hashCode()

↓  
return name.hashCode() + age;  
f.

X ④ public int hashCode()

↓  
return (int)height;

⑤ public int hashCode()

↓  
return age + Name.length();  
f.

Note:-

To maintain a Contract b/w .equals() and hashCode(),

whatever the parameters we are using while overriding

• equals() we have to use the same parameters while overriding  
hashCode() also.

Clone():-

→ The process of Creating exactly duplicate objects is called Cloning

→ The main objective of cloning is to maintain backup.

① We can get cloned object by using clone() of Objects class.

protected native Object clone() throws CloneNotSupportedException

Class Test implements Cloneable

28/1

int i = 10;

int j = 20;

P.S.v.m(---) throws CloneNotSupportedException

↓

Test t<sub>1</sub> = new Test();

Test t<sub>2</sub> = (Test) t<sub>1</sub>.clone();

t<sub>2</sub>.i = 888;

t<sub>2</sub>.j = 999;

S.out(t<sub>1</sub>.i + " --- " + t<sub>1</sub>.j);

S.out(t<sub>1</sub>.hashCode() == t<sub>2</sub>.hashCode()); // false

S.out(t<sub>1</sub> == t<sub>2</sub>); // false.

→ We can call clone() only on cloneable objects.

→ An object is said to be cloneable iff the corresponding class implements  
Cloneable interface. Cloneable interface (presently java.lang package)  
doesn't contain any methods. It is a marker interface.

Deep cloning & shallow cloning:-

→ The process of creating just duplicate reference variable but not  
duplicate object is called shallow cloning.

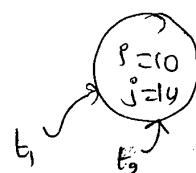
→ The process of creating exactly duplicate independent objects is by  
default considered as deep cloning.

Ex:- Test t<sub>1</sub> = new Test();

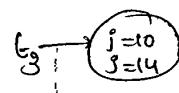
shallow cloning

Test t<sub>2</sub> = t<sub>1</sub>; // shallow cloning

Test t<sub>3</sub> = (Test) t<sub>1</sub>.clone(); // Deep cloning



By default cloning means  
deep cloning.



## String class

24/05/11

Case(1) :-

### Immutable

```
String s = new String("durga");
s.concat(" software");
s.toString(); durga
```



→ Once we created a String object we can't perform any changes in the existing object. If we are trying to perform any changes with those changes a new object will be created. This behaviour is nothing but "immutability of String object".

### mutable

```
SB s = new SB("durga");
s.append(" software");
s.toString(); // durga software
```



→ Once we created a StringBuffer object we can perform any changes in the existing object. This behaviour is nothing but "mutability of StringBuffer object".

getClass() :-

This method returns run-time class definition of an object.

Eg:- Test ob = new Test();

```
ob.toString("Class Name: " + ob.getClass().getName());
```

### Case(2) :-

String  $s_1 = \text{new String}(\text{"durga")};$

String  $s_2 = \text{new String}(\text{"durga")};$

$s_1.o.println(s_1 == s_2); \text{ false}$

$s_1.o.println(s_1.equals(s_2)); \text{ true}$

Stringbuffer  $s_b_1 = \text{new Stringbuffer}(\text{"durga")},$

SB  $s_b_2 = \text{new SB}(\text{"durga")},$

$s_1.o.println(s_b_1 == s_b_2); \text{ false}$

$s_1.o.println(s_b_1.equals(s_b_2)); \text{ false}$

- In String class .equals() method is overridden for Content Comparison.
- Hence .equals() method returns true if Content is same even though Objects are different.

- In Stringbuffer class .equals() method is not overridden for Content Comparison. Hence Object class .equals() method will be executed which is meant for reference comparison due to this .equals() method returns false even though Content is same if Objects are different.

### Case(3) :-

- \* What is the difference b/w following?

Ex:-

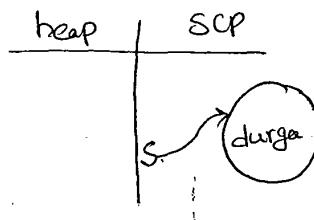
String  $s = \text{new String}(\text{"durga")};$

- In this case two objects will be created one is in heap, & the other is in SCP and 's' is always pointing to heap object



String  $s = \text{"durga"};$

- In this case only one object will be created in SCP and 's' is always pointing to that object



Note:-

- ① GC is not allowed to access in SCP area hence even though Object doesn't have any reference variable still it is not eligible for GC, if it is present in SCP area.
- ② All objects present on SCP will be destroyed automatically at the time of JVM shutdown.
- ③ Object creation in SCP is always optional. First JVM will check is any object already present in SCP with required content or not. If it is already available then it will reuse existing object instead of creating new object. If it is not already available then only a new object will be created. Hence, there is no chance of two objects with the same content in SCP. i.e., duplicate objects are not allowed in SCP.

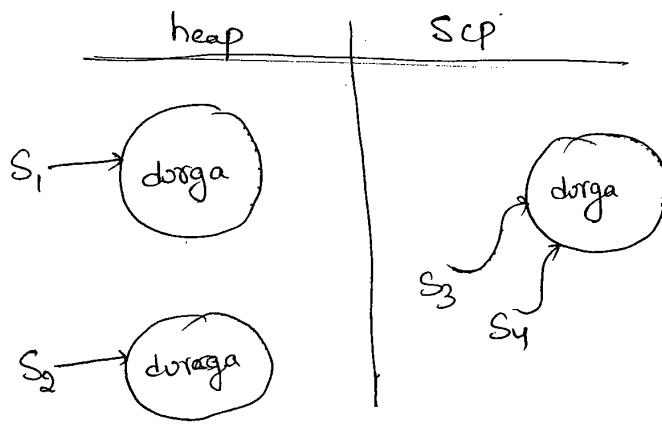
Ex@:

String s<sub>1</sub> = new String ("durga");

String s<sub>2</sub> = new String (" durga");

String s<sub>3</sub> = "durga";

String s<sub>4</sub> = "durga";



Ex(3):-

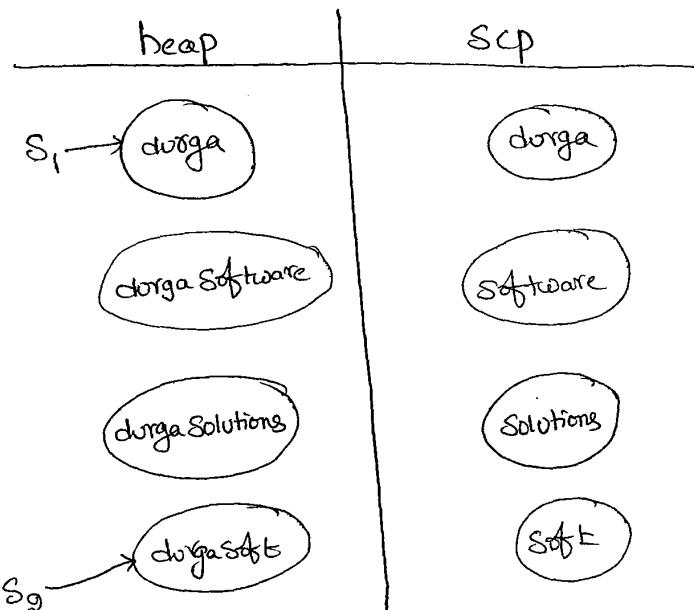
291

String s<sub>1</sub> = new String ("durga");

s<sub>1</sub>. concat ("software");

s<sub>1</sub>. concat ("solutions");

String s<sub>2</sub> = new s<sub>1</sub>. concat ("soft");



Note:-

→ For every String Constant Compulsorily One object will be created in

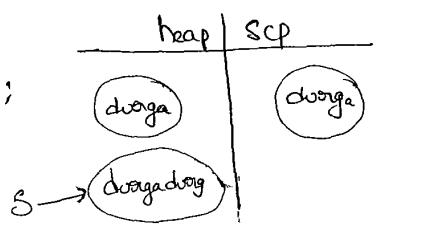
SCP area.

→ Because of some Runtime Operation if an object is required to created

That Object should be Created only on heap but not in SCP

Ex(4):-

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



Ex3:-

String  $s_1 = \text{"Spring";}$

String  $s_2 = s_1 + \text{"Summer";}$

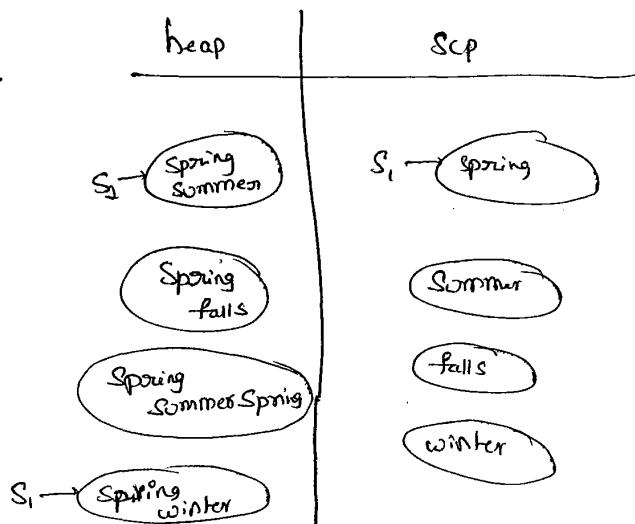
$s_1.\text{Concat}(\text{"falls");}$

$s_2.\text{Concat}(s_1);$

$s_1 += \text{"winter";}$

$s.o.\text{println}(s);$

$s.o.\text{println}(s_2);$



• Expl- Note:-

final String  $s = \text{"raghu";}$   $s$  is a Constant

String  $s = \text{"raghu";}$   $s$  is a normal variable.

Qn:-

String  $s = \text{new String}(\text{"you can't"};$

8/3

992

String  $s_1 = \text{new String("you Cannot change me!");}$

String  $s_2 = \text{new String(" you Cannot change me!");}$

$s_0.\text{println}(s_1 == s_2);$  false

String  $s_3 = "you Cannot change me!";$

String  $s_4 = " You Cannot change me!";$

$s_0.\text{println}(s_1 == s_4);$  true

$s_0.\text{println}(s_1 == s_3);$  false

String  $s_5 = "you Cannot" + "change me!";$

$s_0.\text{println}(s_3 == s_5);$  true

String  $s_6 = "you Cannot";$

String  $s_7 = s_6 + " change me!";$

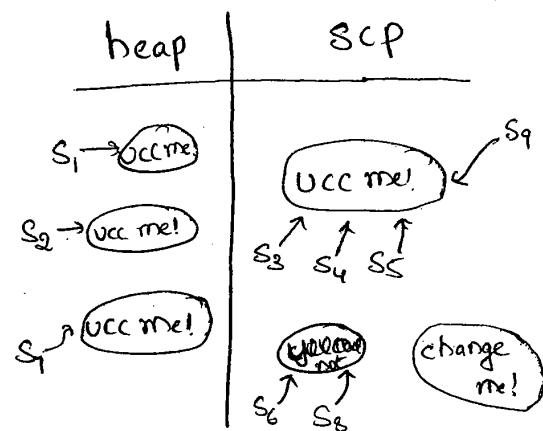
$s_0.\text{println}(s_3 == s_7);$  false

final String  $s_8 = "you Cannot";$

String  $s_9 = s_8 + " change me!";$

$s_0.\text{println}(s_3 == s_9);$  true

$s_0.\text{println}(s_6 == s_8);$  true



### Interning of String :-

→ By using heap object reference if you want to get Correspondingly

SCP object reference then we should go for `intern()`.

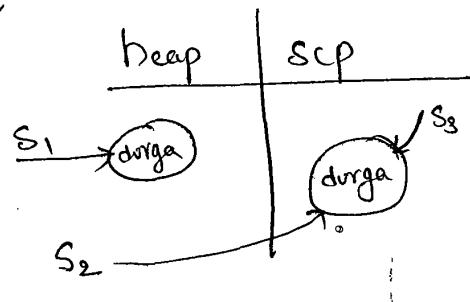
Ex:- String  $s_1 = \text{new String("durga");}$

String  $s_2 = s_1.\text{intern();}$

$s_0.\text{println}(s_1 == s_2);$  false

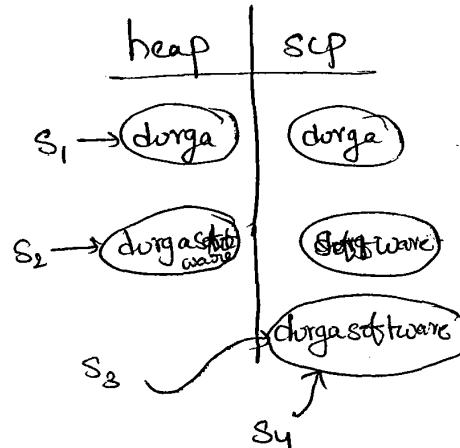
String  $s_3 = "durga";$

$s_0.\text{println}(s_2 == s_3);$  true



→ If the corresponding object not available in SCP, then intern()  
Creates that object & returns it.

Eg:-  
 String s1 = new String("durga");  
 String s2 = s1.concat("software");  
 String s3 = s2.intern();  
 String s4 = "durgaSoftware";  
 S.o.println(s3 == s4); true



### Constructors of the String class:

- ① String s = new String();
- ② String s = new String(String Constant);
- ③ String s = new String(StringBuffer sb);
- ④ String s = new String(char[] ch);

Eg:- char[] ch = {'a', 'b', 'c', 'd'}

String s = new String(ch);  
 S.o.println(s); abcd

- ⑤ String s = new String(byte[] b)

Eg:- byte[] b = {100, 101, 102, 103};

String s = new String(b);  
 S.o.println(s); defg

## Important methods of String class :-

293

① public char charAt (int index);

Eg:- String S = "durga";

S.o.println(S.charAt[3]); g

S.o.println(S.charAt[30]); R.E:- StringIndexOutOfBoundsException

② public String concat (String s);

Eg:- String S = "durga";

S = S.concat ("Software");

// S = S + "Software";

// S != "Software";

S.o.println(S); durgaSoftware

→ The overloaded +, += operators also meant for Concatination Only.

③ public boolean equals (Object obj) meant for Content Comparison

where the Case is also important.

④ public boolean equalsIgnoreCase (String s) meant for Content Comparison

where the Case is not important.

Eg:- String S = "JAVA";

S.o.println (S.equals ("Java")); false

S.o.println (S.equalsIgnoreCase ("java")); true

Note:- In General to perform Validation of User name we have

to go for equalsIgnoreCase method where the Case is not important.

where as to perform password Validation we have to use equals() where the Case is important.

- ⑤ public String substring(int begin); returns the substring from begin index to end of the string.
- ⑥ public String substring(int begin, int end); returns the substring from begin index to end-1 index.

Eg:- String s = "abcdefg";  
 s.println(s.substring(3)); defg  
 s.println(s.substring(2, 6)); cdef

- ⑦ public int length();

Eg:- String s = "aabbbb";  
 s.length(); → C-E: Can't find symbol

✓ s.println(s.length()); s      symbol: variable length  
                                       location: class java.lang.String

#### Note:-

length variable applicable for arrays whereas length() is applicable for string objects.

- ⑧ public String replace(char old, char new);

Eg:- String s = "aabbbb";  
 s.replace('a', 'b'); bbbbb

- ⑨ public String toLowerCase();

- ⑩ public String toUpperCase();

9/3/2011

291

### ① public String trim();

→ To remove the blank spaces present at beginning & end of the String

But not blankspaces present at middle of the String.

### ② public int indexOf(char ch);

→ It returns index of first occurrence of the specified character

### ③ public int lastIndexOf(char ch);

## \* Importance of String Constant pool (Scp):

Voter Registration form

Name of Consistency : cbpet

Name : Srinivas

Fathername: SitaRamaiah

Age : 22

DOB :

H.NO : 9-133

Street : Rammugai

Substreet: Rammugai

City : Ganapavaram

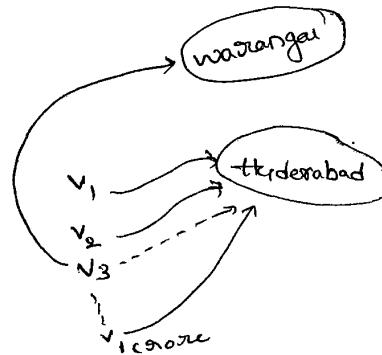
District : Guntur

State : A-p

Country : India

PIN : 522619

Identification Name :  $\begin{matrix} \times \times \times \\ \times \times \times \end{matrix}$



- In our program if any String object required to use repeatedly, it is not recommended to create a separate object for every requirement. This approach reduces performance & memory utilization.
- We can resolve this problem by creating only one object & share the same object with all required references.
- This approach improves memory utilization & performance. We can achieve this by using String Constant pool.
- In SCP, a single object will be shared for all required references. Hence the main advantages of SCP are memory utilization & performance will be improved.
- But the problem in this approach is, As several references pointing to the same object by using one reference, if we are perform any change all remaining references will be impacted.
- To resolve these Sun people declare String objects as immutable.
- According to that once we created a String object we can't perform any change in the existing object. If we are trying to perform any change with  
So, that there is no effect on remaining references
- Hence "the main disadvantage of SCP is we should compulsorily maintain String objects as immutable".

Q) why Scp like Concept is defined only for String object

But not for StringBuffer

A) → In any Java program, The most Commonly used Object is String. Hence with respect to memory & performance Special arrangement is required. For this Scp Concept is required.

→ But StringBuffer is not Commonly used Object. Hence Special Concepts like Scp is not required.

Q) What are the Advantages of Scp?

A) → Instead of Creating a Separate Object for every requirement we can Create Only one object in Scp & we can reuse the same object for Every requirement. So that performance & memory utilization will be increased.

Q) What is the disAdvantage of Scp?

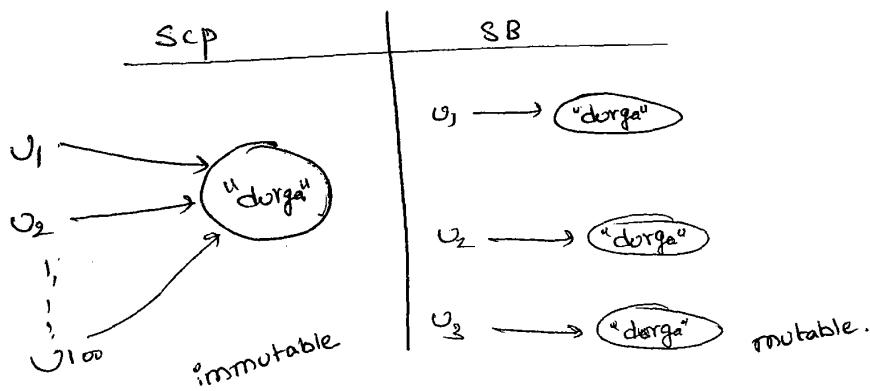
A) → Compulsory we should make String objects as immutable.

Q) Why String objects are immutable whereas StringBuffer Objects are mutable?

A) → In the Case of String Several References can Pointing to the Same object. By using one reference, if we are performing any change in the Existing object The remaining references will be impacted. To resolve this problem SUN people declared as String objects are immutable. According to this Once we created a String object we can't perform any changes in the existing object.

If we are trying to perform any changes, with those changes a new object is created. i.e. SCP is the only reason why the String objects are immutable.

→ But in case of StringBuffer for every requirement Composar a separate object will be created. Reusing the same StringBuffer object, there is no chance. In one StringBuffer object if we are performing any change there is no impact of remaining references. Hence we can perform any changes in the StringBuffer object & StringBuffer objects are mutable.



Q) Is it possible to create our own immutable class?

A) Yes,

Note:

→ Once we created a String object we can't perform any changes in the existing object. If we are trying to perform any change with those changes a new object will be created on the heap.

Heap

→ Because of our runtime method call if there is a change in Content then only new object will be created.

gfb

→ If There is no change in Content Existing object only will be reused.

Ex ① String  $s_1 = "durga";$

String  $s_2 = s_1 \cdot \text{toUpperCase}();$

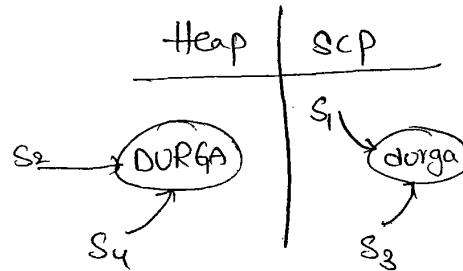
String  $s_3 = s_1 \cdot \text{toLowerCase}();$

String  $s_4 = s_2 \cdot \text{toUpperCase}();$

$s_0 \cdot \text{println}(s_1 == s_2);$  false

$s_0 \cdot \text{println}(s_1 == s_3);$  true

$s_0 \cdot \text{println}(s_2 == s_4);$  true

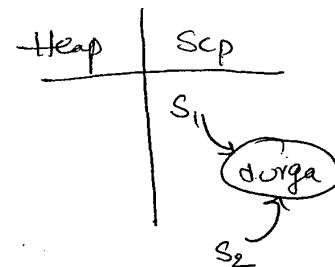


Ex ②:-

String  $s_1 = "durga";$

String  $s_2 = s_1 \cdot \text{toString}();$

$s_0 \cdot \text{println}(s_1 == s_2);$  True



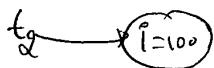
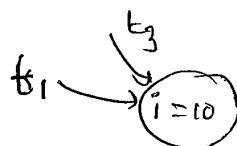
Creation of Our Own Immutable Class :-

We Can Create Our own immutable classes also.

→ Once we Created an object we Can't perform any change in the existing object. If we are trying perform any change with those changes a new object will be Created.

→ Because of our Runtime method Call if There is no change in the Content then Existing object Only will be returned.

Ex :-



Ex:- final class Test

↓

private int i;

Test (int i)

↓

this.i = i;

↳

public Test modify (int i)

↓

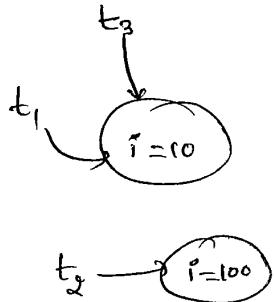
if (this.i == i)

return this;

return (new Test(i)),

↳

↳



Test t<sub>1</sub> = new Test(10);

Test t<sub>2</sub> = new Test(100);

Test t<sub>3</sub> = new Test(10);

S.o.p(t<sub>1</sub> == t<sub>2</sub>); false.

S.o.p(t<sub>1</sub> == t<sub>3</sub>); true



Q) In Java which objects are immutable?

A) (1) String objects ↳

(2) All wrapper objects are immutable

## StringBuffer :-

- If the Content will change frequently then it is never recommended to go for String. Because for every change Compulsory a New Object will be Created.
- To handle this requirement Compulsory we should go for StringBuffer where all changes will be performed in existing object only instead of creating new object.

### Constructors :-

- ① StringBuffer sb = new StringBuffer();
- Creates an Empty StringBuffer object with default initial Capacity 16.
- Once StringBuffer reaches its max. capacity a new SB object will be created with.

$$\text{New Capacity} = (\text{Current Capacity} + 1) * 2$$

### Ex:-

```
StringBuffer sb = new StringBuffer();
```

```
s.o.println(sb.capacity()); // 16
```

```
sb.append("abcdefghijklmnop");
```

```
s.o.println(sb.capacity()); 16
```

```
sb.append("q");
```

```
s.o.println(sb.capacity()); 34.
```

(2) `StringBuffer sb = new StringBuffer(int initialCapacity);`

→ Creates an Empty SB object with specified initialCapacity

(3) `StringBuffer sb = new StringBuffer(String s);`

→ Creates an equivalent SB object for the given String with,

$$\text{Capacity} = 16 + s.length();$$

### Important methods of StringBuffer class:

(1) `public int length()`

(2) `public int capacity()`

(3) `public char charAt(int index);`

Ex: `StringBuffer sb = new StringBuffer("doggo");`

`s.o.println(sb.charAt(3));` g

`s.o.println(sb.charAt(20));` } RE! StringIndexOutOfBoundsException  
`s.o.println(sb.charAt(5));` } Exception.

(4) `public void setCharAt(int index, char ch);`

→ To replace the character Locating at Specified index with the provided Character.

(5) `public StringBuffer append(String s)`

`append (int i)`

`append (boolean b)`  
`(double d)`  
`(Object o)`

} Overloaded methods

Ex:- StringBuffer sb = new StringBuffer();

sb.append("pi value is");

sb.append(3.14);

sb.append("pi is exactly");

sb.append(true);

s.o.println(sb);

⑥ public StringBuffer insert(int index, String s);

(int index, String s);

( " boolean b);

( .11 double d);

}

Ex:- StringBuffer sb = new StringBuffer("durga");

sb.insert(3, "sinu");

s.o.println(sb); dussinuga.

⑦ public StringBuffer delete(int begin, int end);

→ To delete the characters present at begin index to end-1 index

⑧ public StringBuffer deleteCharAt(int index);

→ To delete the character Locating at Specified index.

⑨ public StringBuffer reverse();

Eg:- SB sb = new SB("durga");

s.o.println(sb.reverse()); agurad.

⑩ public void setLength(int length);

⑩ public void setLength(int Length);

Eg:- StringBuffer sb = new StringBuffer("durga123456");  
sb.setLength(8);  
System.out.println(sb); durga123

⑪ public void ensureCapacity(int Capacity);

→ To set the Capacity based on our requirement.

Eg:- StringBuffer sb = new StringBuffer();  
System.out.println(sb.capacity()); 16  
sb.ensureCapacity(2000);  
System.out.println(sb.capacity()); 2000

⑫ public void trimToSize()

→ To release extra allocated free memory. after calling this method, Length & Capacity will be equal.

Eg:- StringBuffer sb = new StringBuffer();  
sb.ensureCapacity(2000);  
sb.append("durga");  
sb.trimToSize();  
System.out.println(sb.capacity()); 5

## StringBuilder :-

299

- Every method present in StringBuffer is Synchronized. Hence at a time only one Thread is allowed to access StringBuffer object.  
It Increases waiting time of the Threads & effects performance of the System.
- To resolve this problem SUN people introduced StringBuilder in 1.5 version.
- StringBuilder is exactly same as StringBuffer (including methods & Constructors) except the following differences:

(8)

| <u>StringBuffer</u>                                                                          | <u>StringBuilder</u>                                                                                 |
|----------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| ① Every method is Synchronized.                                                              | ① No method is Synchronized.                                                                         |
| ② SB object is Thread Safe.<br>Because SB object can be accessed by only one thread at time. | ② StringBuilder is not Thread Safe<br>Because it can be accessed by multiple threads simultaneously. |
| ③ Relatively performance is - Low                                                            | ③ Relatively performance is high.                                                                    |
| ④ Introduced in 1.0 Version                                                                  | ⑤ Introduced in 1.5 Version                                                                          |

## \* String Vs StringBuffer Vs StringBuilder :-

- If the Content <sup>will not</sup> ~~only~~ change frequently Then we should go for String
- If Content will change frequently & ThreadSafety is required. Then we should go for StringBuffer.
- If Content will change frequently & ThreadSafety is not required. Then we should go for StringBuilder.

## Method Chaining:-

- for most of the methods in String, StringBuffer & StringBuilder The return type is same type only. Hence after applying a method on the result we can call another method with forms method chaining

Sb.m<sub>1</sub>().m<sub>2</sub>().m<sub>3</sub>().m<sub>4</sub>().m<sub>5</sub>().....

- In method chaining all methods will be executed from left to right.

Ex:-      StringBuffer    Sb = new StringBuffer();  
              Sb.append("durga").insert(2,"xyz").reverse().del  
                              delete(2,7).append("solutions");  
  
              S.o.println(sb); /agdsolutions

## final vs immutable :-

300

→ If a reference variable declared as the final then we can't reassign that reference variable to some other object.

Ex:-

```
final StringBuffer sb = new StringBuffer("durga");
```

```
sb = new StringBuffer("Software");
```

← E! - Can't assign a value to final variable sb.

→ Declaring a reference variable as final we won't get any immutability nature, in the corresponding object we can perform any type of change even though reference variable declared as final.

Ex:-

```
final StringBuffer sb = new StringBuffer("durga");
```

```
sb.append("Software");
```

```
S.out.println(sb); durgasoftware
```

→ Hence final variable & Immutability both concepts are different.

## \* Wrapper Classes :-

→ The main objectives of wrapper classes are

(i) To wrap primitives into object form, So that we can handle primitives just like objects.

(ii) To define several utility methods for the primitives.

## Constructors of wrapper classes (Q)

### Creation of wrapper objects :-

→ Almost All wrapper classes Contains two Constructors, one can take Corresponding primitive as assignment & The other can take String as assignment.

Ex:- ✓ | Integer I = new Integer(10);

          | Integer I = new Integer("10");

✓ | Double D = new Double(10.5);

          | Double D = new Double("10.5");

→ If the String is not properly formatted Then we will get R.E  
Causing NumberFormatException.

Ex:- Integer I = new Integer("10en"); R.E! - NFE

→ Float class Contains 3 Constructors one can take float primitive,

and the other can take String & 3<sup>rd</sup> one can take double argument

- Ex:
- 1) `float f = new Float(10.5f);` ✓
  - 2) `Float f = new Float("10.5f");` ✓
  - 3) `Float f = new Float(10.5);` ✓ → double

\* Character class Contains only one Constructor which can take char primitive as assignment.

- Ex:-
- i) `Character ch = new Character('a');` ✓
  - ii) `Character ch = new Character("a");` X

\* Boolean class Contains two Constructors one can take Boolean primitive as the assignment & other can take String as assignment.

→ If we are passing boolean primitive as assignment the only allowed values are true, false. by mistake if we are providing any other we will get CompiletimeError.

- Ex:-
- ✓ `Boolean B = new Boolean(true);`
  - X `Boolean B = new Boolean(Ttrue);`
- If we are passing String assignment to the Boolean Constructor then the case is not important & Content also not important.
- If the Content Case insensitive String ~~(true)~~, otherwise it is treated as false.

- Ex:-
- (1) `Boolean b = new Boolean("true");` ✓ true
  - (2) `Boolean b = new Boolean("True");` ✓ true
  - (3) `Boolean b = new Boolean("TRUE");` ✓ true
  - (4) `Boolean b = new Boolean("durga");` ✓ false
  - (5) `Boolean b = new Boolean("yes");` ✓ false

## Wrapper classes

## Corresponding Constructor arrangement

|             |                           |
|-------------|---------------------------|
| Byte        | byte or String            |
| Short       | short or String           |
| Integer     | int or String             |
| Long        | long or String            |
| * Float     | float or String or double |
| Double      | double or String          |
| * Character | char                      |
| * Boolean   | boolean or String         |

Q): Which one is True & False

(1) Boolean b<sub>1</sub> = new Boolean("yes");

(2) Boolean b<sub>2</sub> = new Boolean("no");

S.o.println(b<sub>1</sub>.equals(b<sub>2</sub>)); → true

S.o.println(b<sub>1</sub> == b<sub>2</sub>); → false

S.o.println(b<sub>1</sub>); false

S.o.println(b<sub>2</sub>); false.

Note:-

→ In Every wrapper class `toString()` is overridden to return its Content.

→ In Every wrapper Class `equals()` is overridden to Content Comparison.

### Utility Methods :-

There are 4 methods

(i) `valueOf()`

(ii) `xxxValue()`

(iii) `parseXXX()`

(iv) `toString()`

#### (i) valueOf() :-

methods

→ We can use `valueOf()`, for Creating wrapper object as Alternative to Constructor.

#### Form :-

→ Every wrapper class Except Character Class Contains a Static `valueOf()` method for Converting for Converting String to the wrapper Object.

Public static wrapper `valueOf(String s)`

Eg:- `Integer I1 = Integer.valueOf("10");` ✓

`Boolean b1 = Boolean.valueOf("true");` ✓

`Double D = Double.valueOf("10.5");` ✓

### Form (2) :-

→ Every Integral type wrapper class (Byte, Short, Integer, Long) Contains the following valueOf() method → to Convert Specified Radix String form to Corresponding Wrapper object.

```
public static wrapper valueOf(String s, int radix);
```

Eg:-

Integer I<sub>1</sub> = Integer.valueOf("1010", 2);

S.o.println(I<sub>1</sub>); 10

2 to 36

base-10 : 0-9

base-11 : 0-9, A

base-16 : 0-9, A-F

base-17 : 0-9, A-G

base-36 : 0-9, A-Z

10 + 26

= 36

Integer I<sub>2</sub> = Integer.valueOf("1111", 2);

S.o.println(I<sub>2</sub>); 15

### Form (3) :-

→ Every wrapper class including Character class Contains the following valueOf() to Convert primitive to Corresponding wrapper Object

```
public static wrapper valueOf(primitive p);
```

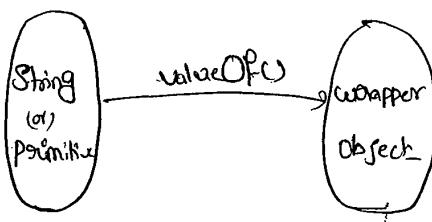
Eg:-

1) Integer I = Integer.valueOf(10); ✓

2) Character ch = Character.valueOf('a'); ✓

3) Boolean B = Boolean.valueOf(true); ✓

Note:-



15/03/11

(ii) xxxValue() :-

302

→ We can use xxxValue() methods to convert wrapper object to primitive.

→ Every Number type wrapper class contains the following six(6) xxxValue() methods.

→ The methods are

```

public byte byteValue();
public int intValue();
public short shortValue();
public long longValue();
public float floatValue();
public double doubleValue();

```

e.g.:-

```

(1) Double D = new Double(130.456);
 S.o.println(D.byteValue()); -126
 S.o.println(D.shortValue()); 130
 S.o.println(D.intValue()); 130
 S.o.println(D.longValue()); 130
 S.o.println(D.floatValue()); 130.0
 S.o.println(D.doubleValue()); 130.0

```

charValue() :-

→ Character class contains charValue method to convert Character object to the ~~char~~ char primitive.

→ Public char charValue();

Eg:- Character ch = new Character('@');  
 char ch1 = ch.charAt();  
 System.out.println(ch1); '@'

booleanValue()!

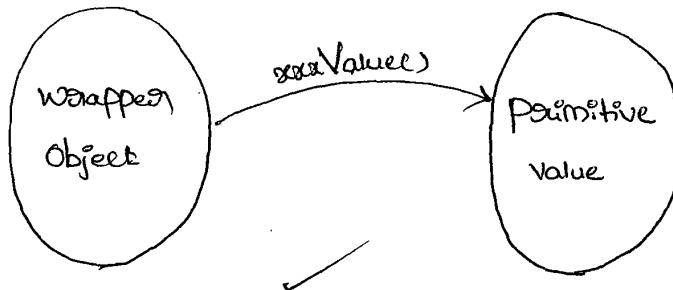
→ Boolean Class Contains booleanValue() to find boolean primitive for the given boolean Object.

public boolean booleanValue();

Eg:- Boolean B = Boolean.valueOf("duong");  
 boolean b = B.booleanValue();  
 System.out.println(b); false.

Note:-

→ Total 38 ( $6 \times 6 + 1 + 1$ ) xxxValue() are available.



(Q1) parseXXX() :-

303

→ We can use `parseXXX()` to Convert String to Corresponding Primitive.

Form1 :-

→ Every Wrapper class Except Char Class Contains the following `parseXXX()` to, Convert String to Corresponding Primitive.

public static primitive parseXXX(String s);

Eg:-

int i = Integer.parseInt("10");

double d = Double.parseDouble("10.5");

long l = Long.parseLong("10L");

Boolean b = Boolean.parseBoolean("durga"); op- false

Form2 :-

→ Every Integral-type Wrapped class Contains the following `parseXXX()` to Convert Specified Radix String to Corresponding primitive.

Eg:-

public static primitive parseXXX(String s, int radix);

Eg:-

int i = Integer.parseInt("1111", 2);

So, `System.out.println(i);` ; 15

2 to 36.

Note:-



(iv) `toString()` :-

→ We can use `toString()` to Convert Wrapper Object or Primitive to String.

`format()` :-

→ Every wrapper class Contains The following `toString()`, ~~to~~ to Convert Wrapper Object to String type.

public String toString();

→ It is the Overriding Version of Object class `toString()`.

Eg:-

① Integer I = New Integer(10);  
S.out(I.toString()); 10 ✓

`format(2)` :-

→ Every wrapped class Contains a Static `toString()`, to Convert primitive to String form.

public static String toString(primitive p);

✓ String s = Integer.toString(10);

✓ String s = Boolean.toString(true);

`format(3)` :-

→ Integer & Long classes Contains `toString()` to Convert primitive to Specified Radix String form.

public static String toString(primitive p, int radix); } 2 to 36

Eg. ✓ String s = Integer.toString(15, 2);

s.o. println(s); 1111

From :-

→ Integer & Long classes Contains The following toXXXString()

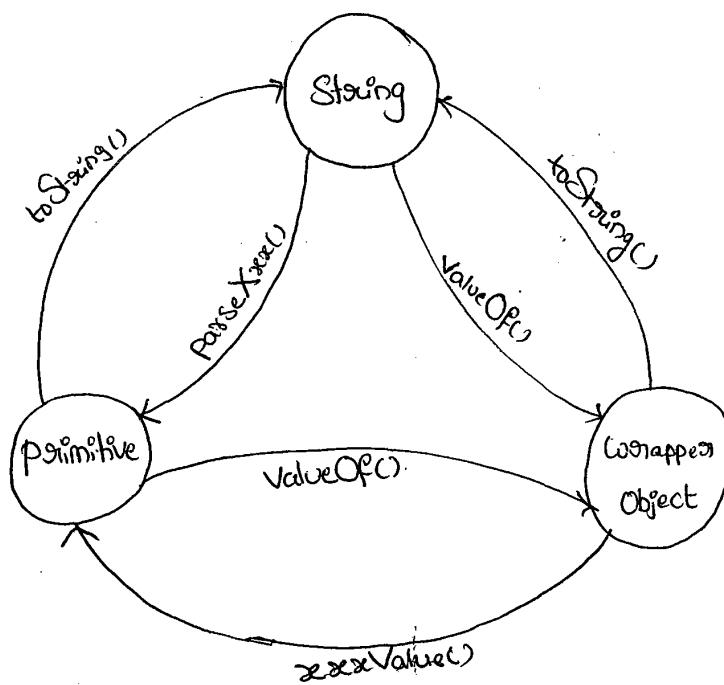
1. public static String toBinaryString(primitive p);
2. public static String toOctalString(primitive p);
3. public static String toHexString(primitive p);

Ex. ✓ String s = Integer.toHexString(123)

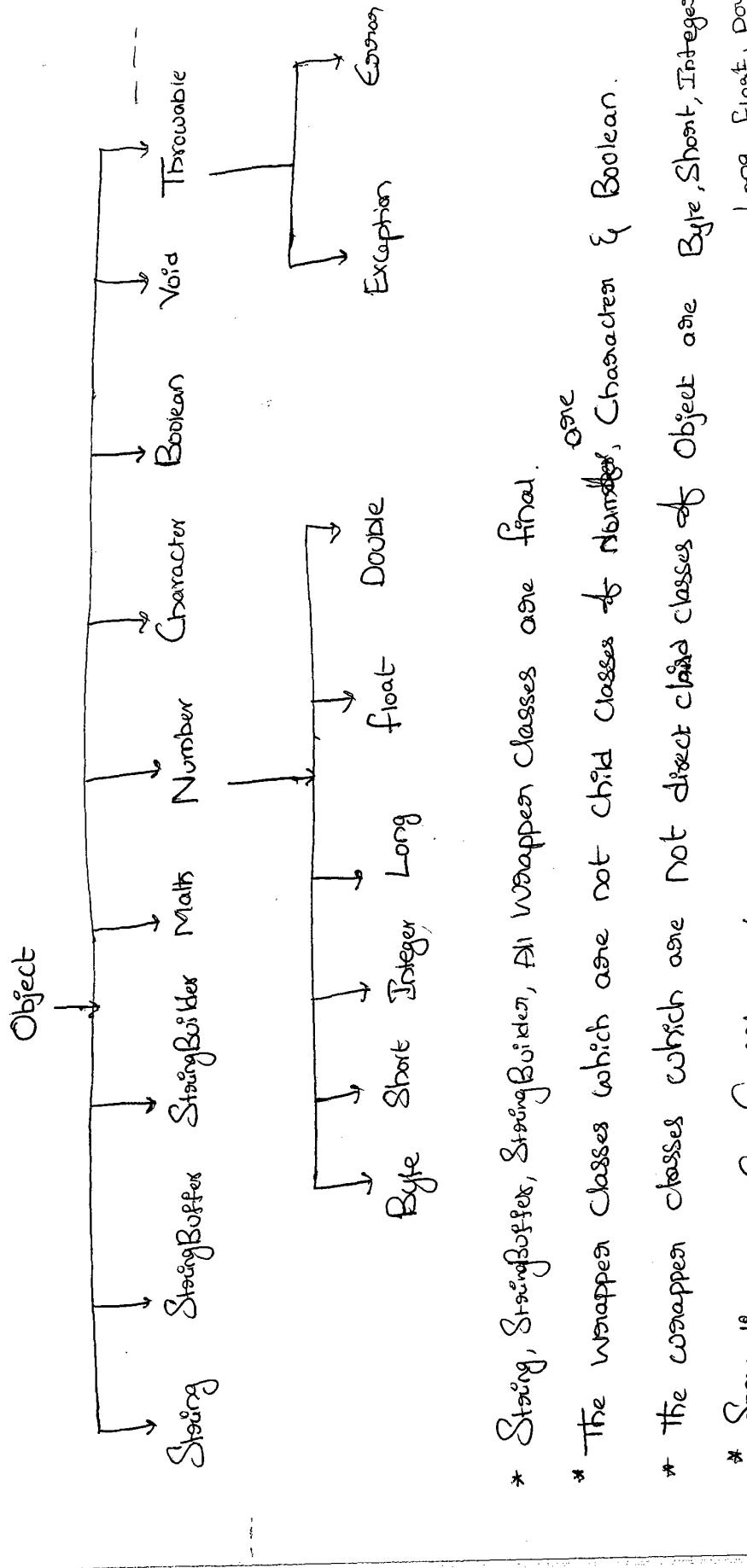
✓ s.o. println(s); "7b"

16 | 123  
      |  
      7 - b

Dancing b/w String, wrapped Object, & primitive Value:-



## Partial hierarchy of java.lang package:-



- \* String, StringTokenizer, StringBuilder, All Wrapper Classes are final.
- \* The wrapper classes which are not child classes to numbers, Character & Boolean.
- \* The wrapper classes which are not direct child classes of object are Byte, Short, Integer, Long, Float, Double.
- \* Sometimes we can consider Void also as wrapper Classes
- \* In addition to String object all wrapper objects are Immutable.

16-3-11

## Auto boxing & Auto unboxing :-

(1.5v)

305

→ until 1.4 version we can't provide primitive value in the place of wrapper objects & wrapper objects in the place of primitive. All the required conversions should be performed explicitly by the programmer.

Ex:-

① ArrayList l = new ArrayList();

l.add(10); X C.E!

② Integer I = new Integer(10);

l.add(I); ✓

③ Boolean B = new Boolean(true);

if(B)

C.E! -

S.o.println("Hello");

Incompatible types

found : Boolean

required : boolean

boolean b = B.booleanValue();

if(b) —

{  
S.o.println("Hello");

✓

→ But from 1.5 version onwards in the place of wrapper objects we can provide primitive value & in the place of primitive value we can provide wrapper objects. All the required conversions will be performed automatically by the compiler. These automatic

Conversions are called Autoboxing & Auto unboxing.

Autoboxing:-

→ Automatic Conversion of primitive value to the wrapper object by Compiler is called "Autoboxing".

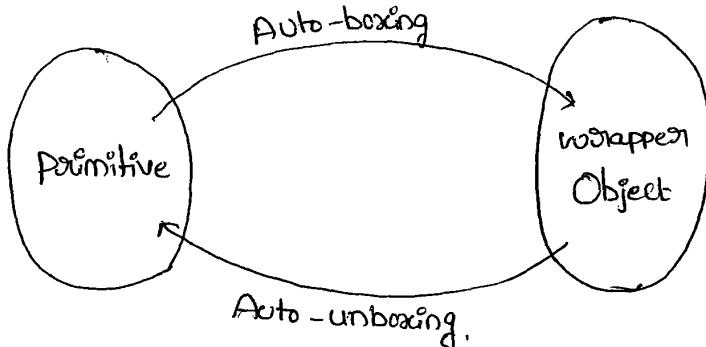
Ex:- ✓ `Integer I = 10;` [Compiler Converts `int` to `Integer` automatically by Autoboxing]

Auto-unboxing:-

→ Automatic Conversion of wrapper Object to the primitive type by Compiler is called "Auto-unboxing".

Ex:- ✓ `int i = new Integer(10);` [Compiler Converts `Integer` to `int` automatically by Auto-unboxing]

Note:-



Ex:- ① `Integer I = 10;`

↳ after Compilation This Line will become

`Integer I = Integer.valueOf(10);`

i.e., Autoboxing Concept internally implemented by using valueOf()

Ex@:-

Integer I = new Integer(10);

int i = I;

→ After Compilation this Line will become

int i = I.intValue();

i.e., Autounboxing Concept internally implemented by using xxxValue().

Exam purpose:-

Ex@:-

Class Test

↓

Static Integer I = 10; → ① A.B

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

↓

int i = I; → ② A.U.B

m1(i); → ③ A.B

P.S.V.m1 (Integer I)

↓

int k = I; → ④ A.Q.B

S.O.Println(k); 10

↓

Note:-

→ Because of Autoboxing & Auto-unboxing, from 1.5 Version onwards

There is no diff. b/w primitive Value & Wrapper Object. we can use interchangeable.

Ex 2 :-

```

class Test
{
 static Integer I=0;
 P.S.V.m(String[] args)
 {
 int i = I;
 S.o.println(i); //0
 }
 int i = I.intValue();
}

```

```

class Test
{
 static Integer I;
 P.S.V.m(String[] args)
 {
 int i = I; → R.E:- NPE
 S.o.println(i);
 }
 int i = I.intValue(); ← Null
}

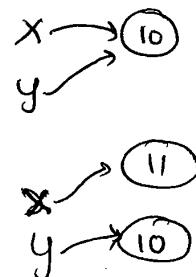
```

Ex 3 :-

```

Integer x = 10;
Integer y = x;
x++;
✓ S.o.println(x); 11
✓ S.o.println(y); 10
✓ S.o.println(x==y); false

```



Note :-  
because if we want to change after creating an object, then that new changed object is created with the same reference name.

Ex 4 :-

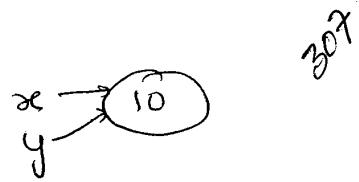
① Integer X = new Integer(10);  
 Integer Y = new Integer(10);  
 S.o.println(x==y); false ✓

② Integer x = new Integer(10);  
 Integer y = (0);  
 S.o.println(x==y); false ✓

③ Integer x = 10;

Integer y = 10;

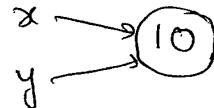
S.o.println(x == y); true ✓



④ Integer x = 100;

Integer y = 100;

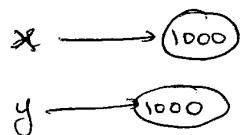
S.o.println(x == y); true ✓



⑤ Integer x = 1000;

Integer y = 1000;

S.o.println(x == y); false ✓



### Conclusion :-

- By AutoBoxing if an Object is required to Create Compiler won't
- Create that object immediately. First check is any object already created
- If it is already Created then it will reuse existing object. instead of creating new one.
- If it is not already there, then only a new object will be created.
- But this rule is applicable only in the following cases,

① Byte → Always

② Short → -128 to 127

③ Integer → -128 to 127

④ Long → -128 to 127

⑤ Character → 0 to 127

⑥ Boolean → Always

→ Except the above range in all other cases Compulsorily a new object - will be created.

Ex:-

- ① Integer  $I_1 = 127;$   
Integer  $I_2 = 127;$   
`S.o.println(I1 == I2);` true
- ② Integer  $I_1 = 128;$   
Integer  $I_2 = 128;$   
`S.o.println(I1 == I2);` false
- ③ Float  $f_1 = 10.0f;$   
Float  $f_2 = 10.0f;$   
`S.o.println(f1 == f2);` false
- ④ Boolean  $b_1 = \text{true};$   
Boolean  $b_2 = \text{true};$   
`S.o.println(b1 == b2);` true.

→ Overloading w.r.t auto-boxing, widening & Var-Arg methods.

case(1) :-

widening Vs Auto-boxing :-

Ex:- Class Test

```

 p.s.v.m1(long l)
 ↓
 S.o.println("widening");
 ↓
 p.s.v.m2(Integer I)
 ↓
 S.o.println("Autoboxing");
 ↓

```

- ① Byte → Always
- ② Short → -128 to 127
- ③ Integer → -128 to 127
- ④ Long → -128 to 127
- ⑤ Character → 0 to 127
- ⑥ Boolean → Always

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

308

{

int x = 10;

m1(x); op!-  
widening

}

}

→ widening dominates Auto-boxing

Case(2):-

→ Widening Vs Var-args.

) Op!- Class Test

{

P.S.V.m1(long l)

{

S.o.println("widening");

}

P.S.V.m1(int... i)

{

S.o.println("Var-args");

}

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

{

int x = 10;

m1(x);

op!- widening

}

→ widening dominates Var-args.

### Case 3:-

→ Auto-boxing Vs Var-arg :-

Ex:- Class Test

```
p.s.r.m1(Integer I)
{
 System.out.println("Auto boxing");
}
p.s.r.m1(int... i)
{
 System.out.println("Var-arg");
}
p.s.r.m1(String[] args)
{
 int x=10;
 m1(x); off:- Autoboxing.
}
```

→ In General Var-arg() will get least priority, if no other method matched then only Var-arg() will be executed.

→ while Resolving overloaded methods Compiler will always keeps the precedence in the following order.

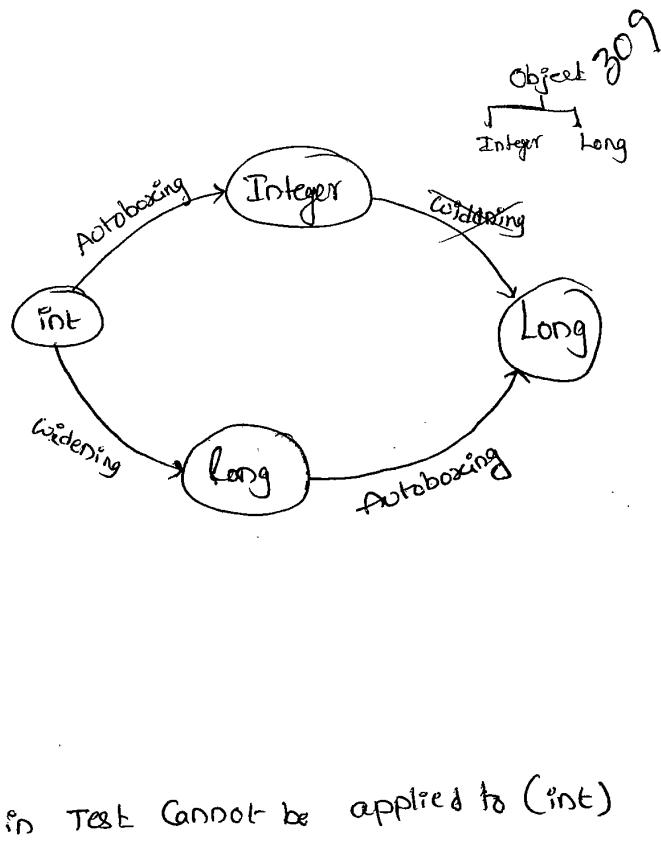
- (i) Widening
- (ii) Auto-boxing
- (iii) Var-arg().

Case 4 :-

```

class Test
{
 p.s.v.m1(Long l)
 {
 S.o.println("Long");
 }
 p.s.v.main(String[] args)
 {
 int x=10;
 m1(x);
 }
}

```



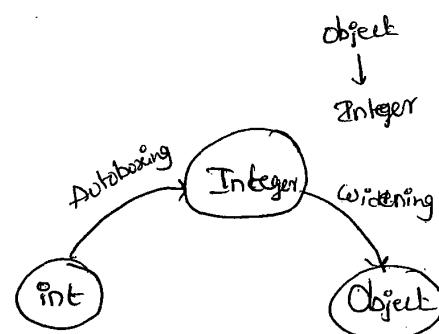
m1(java.lang.Long) in Test Cannot be applied to (int)

- Widening followed by Auto-boxing is not allowed in java. whereas
- Auto-boxing followed by Widening is allowed.

```

ex:- class Test
{
 p.s.void m1(Object o)
 {
 S.println("Object");
 }
 p.s.void. main(String[] args)
 {
 int x=10;
 m1(x); as Object ✓
 }
}

```



Q) Which of the following declarations are Valid.

✓ ① long l = 10;

✗ ② Long l = 10;

✓ ③ Object o = 10;

✓ ④ double d = 10;

✗ ⑤ Double d = 10;

✓ ⑥ Number n = 10;



وَلِلَّهِ الْحُكْمُ وَإِلَيْهِ الْمُرْجَعُ فَلَا يُنَزَّلُ مِنْهُ إِلَّا مُحْكَمٌ

وَلِلَّهِ الْحُكْمُ وَإِلَيْهِ الْمُرْجَعُ فَلَا يُنَزَّلُ مِنْهُ إِلَّا مُحْكَمٌ

05/04/11

## Java.io package

3/2

### File I/O :-

1. file
2. FileWriter
3. FileReader
4. BufferedReader
5. BufferedWriter
6. PrintWriter.

#### (1) file :-

```
File f = new File("abc.txt");
```

→ This line won't create any physical file, first it will check is there any file named with abc.txt is available or not.

→ If it is available then f simply pointing to that file.

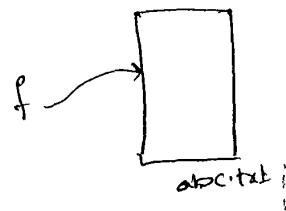
→ If it is not available then f represents just name of the file without creating any physical file.

```
File f = new File("abc.txt");
```

```
S.o.println(f.exists()); // false
```

```
f.createNewFile();
```

```
S.o.println(f.exists()); // true.
```



→ A Java file object can represent a directory also.

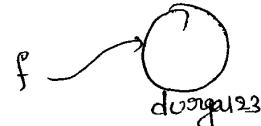
Ex:-

```
file f = new File("durga23");
```

```
sop(f.exists()); false
```

```
f.mkdir();
```

```
sop(f.exists()); true
```



### Constructors:-

① file f = new File(String name);

→ Create a java file object to represent name of a file or directory.

② file f = new File(String Subdir, String name);

→ To Create a file or directory present in Some other Sub-directory.

③ file f = new File(File Subdir, String name);

Ex:- Write Code to Create a file named with abc.txt in Current Working directory.

```
file f = new File("abc.txt");
```

```
f.createNewFile();
```

④ w.c. to Create a directory named with xyz in Current working directory.

```
file f = new File("xyz");
```

```
f.mkdir();
```

#### (4) flush() :-

→ to give the guarantee that last character of the data also refers to the file.

#### (5) close() :-

Ex:- demo program for the FileWriter.

```
import java.io.*;
class FileDemo2
{
 public static void main(String[] args)
 {
 FileWriter fw = new FileWriter("wc.txt", true);
 fw.write('a'); // adding a single character
 fw.write("vagin softwareSolutions");
 char[] ch1 = {'a', 'b', 'c'};
 fw.write('m');
 fw.write(ch1);
 fw.write('n');
 fw.flush();
 fw.close();
 }
}
```

Appending

Op:- d  
vagin  
SoftwareSolutions  
abc

100 → d

## (2) FileWriter :-

→ We can use `FileWriter` object to write character data to the file.

### Constructors :-

① `fileWriter fw = new fileWriter(String name);`

② `fileWriter fw = new fileWriter(File f);`

→ The above 2 Constructors meant for overriding. If we want to perform append instead of overriding then we have to use the following Constructors.

③ `fileWriter fw = new fileWriter(String name, boolean append);`

④ `fileWriter fw = new fileWriter(File f, boolean append);`

→ If the Specified file is not already available then the above Constructors will Create that file.

### Methods of fileWriter :-

① `write(int ch);`

To write a Single character to the file.

② `write(char[] ch);`

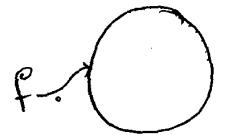
To write an array of characters to the file.

③ `write(String s);`

To write a String to the file.

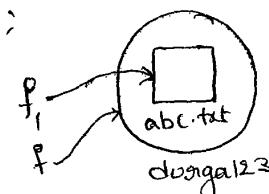
③ w.c. to Create a directory named with durga123 in current working directory. in that directory create a file named with abc.txt.

A) `file f = new File("durga123");  
f.mkdir();`



`file f1 = new File("durga123", "abc.txt");  
f1.createNewFile();`

(a)



`file f1 = new File( f, "abc.txt");  
f1.createNewFile();`

### Important methods of File class :-

① `boolean exists();`

→ returns true if the Physical file or directory present

② `boolean CreateNewFile();`

→ first this method will check whether the Specified file is already available or not. If it is already available then this method won't return false without creating newfile. If it is not already available then this method returns true after creating newfile.

③ `boolean mkdir();`

④ `boolean isFile();`

(5) boolean isDirectory();

(6) String[] list();

→ It returns the names of all files & Sub-directories present  
in the Specified directory.

(7) boolean delete();

→ To delete a file or directory

(8) long length();

→ Returns the no. of characters present in the Specified-  
file

Ex:- W.a.p to print the names of all files & sub-directories  
present in "D:\durga-classes".

```
import java.io.*;
class Test
{
 public static void main(String[] args) throws Exception
 {
 File f = new File("D:\\durga-classes");
 String[] s = f.list();
 for (String s1 : s)
 {
 System.out.println(s1);
 }
 }
}
```

### (3) FileReader :-

→ we can use `FileReader` to Read character data from the file

#### Constructors :-

1. `FileReader fr = new FileReader(String name);`
2. `FileReader fr = new FileReader(File f);`

#### Methods of FileReader :-

##### (i) int read(); :-

\* It attempts to read next character from the file and return its Unicode value.

\* If the next character is not available, then this method returns `-1`.

##### (ii) int read(char[] ch); :-

\* It attempts to read enough characters from the file into the char array & returns the no. of characters which are copied from file to the `char[]`.

##### (iii) close();

#### Ex:- on FileReader

```
import java.io.*;
```

```
Class fileReadDemo
```

```
{
```

```
P-S-V-m (String[] args) throws IOException,
```

```
file f = new file ("wc.txt")
fileReader fr = new FileReader(f);
System.out.println(fr.read()); // Unicode of first character
char[] ch2 = new char[(int)f.length()];
fr.read(ch2); // file data Copied to array
for (char ch : ch2)
{
 System.out.print(ch);
}
System.out.println("-----");
fileReader fr1 = new FileReader(f);
int i = fr1.read();
while (i != -1)
{
 System.out.println((char)i);
 i = fr1.read();
}
```

\* Usage of FileWriter & FileReader is not recommended because :-

- 1) while writing data by `fileWriter` we have to insert line separators manually which is a bigger headache to the programmer.
- 2) By using `fileReader` we can read data character by character which is not convenient ~~to the~~ programmer.
- 3) To resolve these problem SUN people introduced `BufferedWriter` & `BufferedReader` classes.

(ii) `BufferedWriter` :-

→ We can use `BufferedWriter` to write character data to the file.

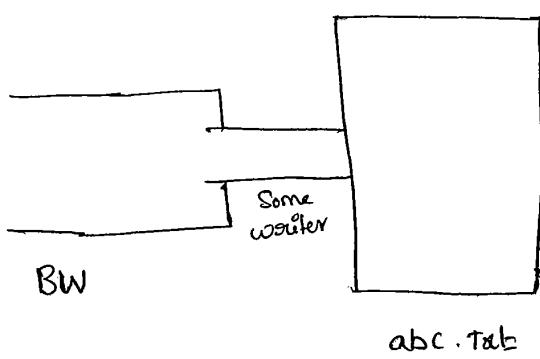
Constructors :-

1) `BufferedWriter bw = new BufferedWriter(Writer w);`

2) `BufferedWriter bw = new BufferedWriter(Writer w, int bufferSize);`

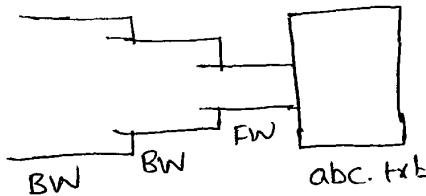
Note!

→ `BufferedWriter` never communicates directly with the file Compulsory it should communicate via some writer object only.



Q) which of the following are valid.

- ① `BufferedWriter bw = new BufferedWriter("abc.txt");`
- ② `BufferedWriter bw = new BW(new file("abc.txt"));`
- ③ `BufferedWriter bw = new BufferedWriter(new FileWriter("abc.txt"));`
- ④ `BW bw = new BW(new BW(new FW(new file("abc.txt"))));`



Important methods of BufferedWriter :-

- ① `write(int b)`
- ② `write(char[] ch)`
- ③ `write(String s)`
- ④ `flush()`
- ⑤ `close()`
- ⑥ `newLine();` :- to insert a newline character

Q:- When Compared with FileWriter which of the following Capability is available as a Separate method in BufferedWriter.

- A(1)
- ① Writing data to the file.
  - ② flushing the Stream.
  - ③ closing the Stream.
  - ④ inserting a line separator.

Ex:-

```

import java.io.*;
class BufferedWriterDemo
{
 public static void main(String[] args) throws IOException
 {
 File f = new File("wc.txt");
 FileWriter fw = new FileWriter(f);
 BufferedWriter bw = new BufferedWriter(fw);
 bw.write(100);
 bw.newLine();
 char[] chs = {'a','b','c','d'};
 bw.write(chs);
 bw.newLine();
 bw.write("duaga");
 bw.newLine();
 bw.write("Software Solutions");
 bw.flush();
 bw.close();
 }
}

```

O/P:-

|                    |
|--------------------|
| d                  |
| abcd               |
| duaga              |
| Software Solutions |
| wc.txt             |

Note:- When ever we are closing BufferedWriter automatically underlying Writers will be closed

|             |   |             |  |             |
|-------------|---|-------------|--|-------------|
| BW.close(); |   | fw.close(); |  | fw.close(); |
| ✓           | X |             |  | X           |

#### (iv) BufferedReader :-

- \* The main advantage of BufferedReader over FileReader is we can read the data line by line instead of reading character by character. This approach improves performance of the system by reducing the no. of read operations.

#### Constructors :-

(i) BufferedReader br = new BufferedReader(Reader r);

(ii) " " " (Reader r, int bufferSize);

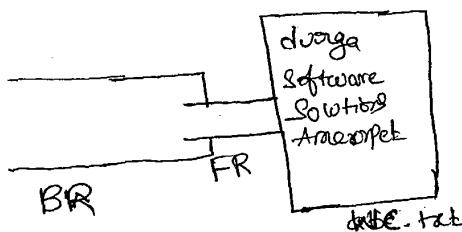
#### Note:-

- \* BufferedReader can't communicate directly with the file. It should communicate via some Reader object.

#### Important Methods :-

- ① int read();
- ② int read(char[] ch);
- ③ close();
- ④ String readLine();

- \* It attempts to find the next line & if the next line is available then it returns it, otherwise it returns null.



```

Ex:- import java.io.*;

class Buffered
{
 public static void main(String[] args) throws Exception
 {
 FileReader fr = new FileReader("wc.txt");
 BufferedReader br = new BufferedReader(fr);
 String line = br.readLine();
 while(line != null)
 {
 System.out.println(line);
 line = br.readLine();
 }
 br.close();
 }
}

```

O/P:-

```

durga
Software
Solutions
Amreepet

```

Note:-

- \* whenever you're closing BufferedReader (underlying Readers will be closed).

## V) PointWriter :-

\* This is the most enhanced writer to write character data to file. By using FileWriter & BufferedWriter we can write only character data but by using PointWriter we can write any primitive data-types to the file.

### Constructors:-

- ① PointWriter pw = new PointWriter(String name);
- ② PointWriter pw = new PointWriter(File f);
- ③ PointWriter pw = new PointWriter(Writer w);

### Methods:-

|                    |                  |                    |
|--------------------|------------------|--------------------|
| ① write(int ch)    | Point(char ch)   | pointIn(char ch)   |
| ② write(char[] ch) | point(int i)     | pointIn(int i)     |
| ③ write(String s)  | point(long l)    | pointIn(long l)    |
| ④ flush()          | point(double d)  | pointIn(double d)  |
| ⑤ close()          | point(String s)  | pointIn(String s)  |
|                    | point(char[] ch) | pointIn(char[] ch) |
|                    | !                | !                  |

Ex:-

```
import java.io.*;
Class PointWriterDemo1
{
 p. s. v. m (String[] args) throws IOException
```

3/9

```
FileWriter fw = new FileWriter("wc.txt");
```

```
PrintWriter pw = new PrintWriter(fw);
```

```
pw.write(100); // 100
```

```
pw.println(100); // 100
```

```
pw.println(true); // true
```

```
pw.println('c'); // c
```

```
pw.println("durga"); // durga
```

```
pw.flush();
```

```
pw.close();
```

O/P:-

d100  
true  
c  
durga

wc.txt

Q1: what is the diff. b/w the following

(a) pw.write(100);

(b) pw.print(100);

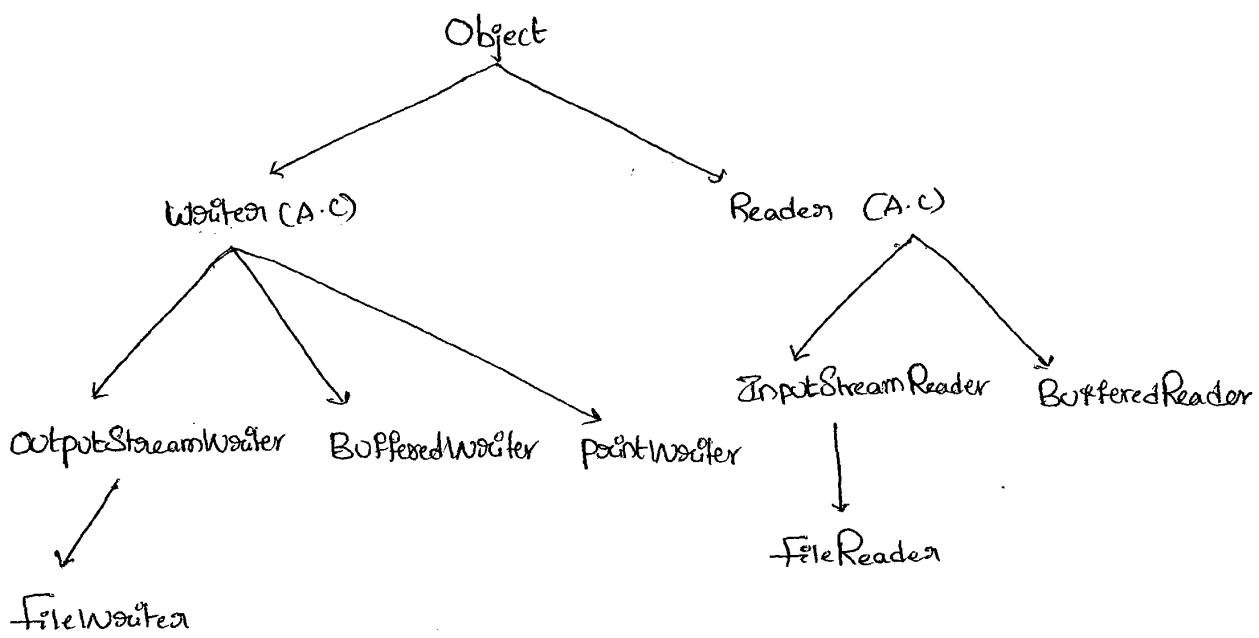
Ans:-

Pw.write(100);

→ In this case the corresponding character d will be added to the file

Pw.print(100)

→ In this Case the Corresponding int value 100 directly will be added to the file



#### Note :-

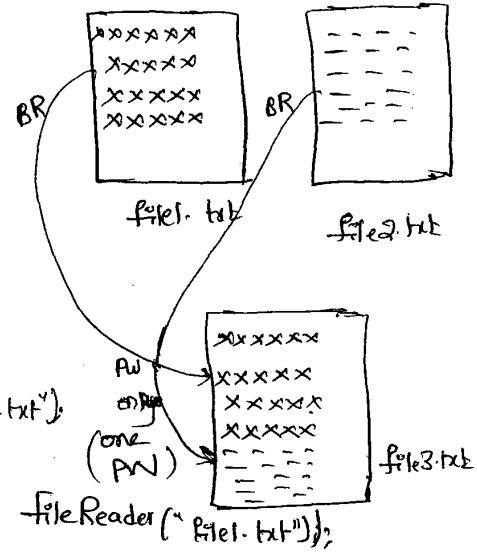
- \* Readers & writers meant for handling character data (any primitive data type) +
- \* To handle Binary data (like images, movie files, jar files ....) we Should go for Streams.
- \* We can use **InputStream** to Read Binary data & **OutputStream** to write a Binary data.
- \* We can use **ObjectInputStream** & **ObjectOutputStream** to read & write Objects to a file respectively (Serialization).
- \* The most Enhanced writer to write character data is **PrintWriter** whereas the most Enhanced Reader to read character data is **BufferedReader**.

\*) W.a.p to merge data from two files into a 3<sup>rd</sup> file. 32

file3.txt = file1.txt + file2.txt

```
import java.io.*;
class FileMerger
{
 public static void main(String[] args) throws Exception
```

```
 PrintWriter pw = new PrintWriter("output.txt");
 BufferedReader br1 = new BufferedReader(new FileReader("file1.txt"));
 String line = br1.readLine();
 while (line != null)
 {
 pw.println(line);
 line = br1.readLine();
 }
 br1 = new BufferedReader(new FileReader("file2.txt"));
 line = br1.readLine();
 while (line != null)
 {
 pw.println(line);
 line = br1.readLine();
 }
 pw.flush();
 br1.close();
 pw.close();
}
```



Ex2:-

w.a.p to merge data from 2 files into a 3<sup>rd</sup> file but merging  
should be done line by line alternatively.

```
import java.io.*;
```

```
class fileMerger2
```

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

```
{
```

```
PointWriter pw = new PointWriter("output.txt");
```

```
BufferedReader bsr1 = new BufferedReader(new FileReader("file1.txt"));
```

```
BufferedReader bsr2 = new BufferedReader(new FileReader("file2.txt"));
```

```
String line1 = bsr1.readLine();
```

```
String line2 = bsr2.readLine();
```

```
while ((line1 != null) | (line2 != null))
```

```
{
```

```
if (line1 != null)
```

```
{
```

```
pw.println(line1);
```

```
line1 = bsr1.readLine();
```

```
}
```

```
if (line2 != null)
```

```
{
```

```
pw.println(line2);
```

```
line2 = bsr2.readLine();
```

```
}
```

```
pw.flush();
```

```
pw.close();
```

```
bsr1.close();
```

```
bsr2.close();
```

```
}
```

```
if(available == false)
```

```
{
 pw.println(line);
```

```
 pw.flush();
```

```
}
```

```
line = bai.readLine();
```

```
{
```

```
 pw.flush();
```

```
 bai.close();
```

```
 bai.close();
```

```
 pw.close();
```

```
}
```

④ W.A.P to perform File Extraction (result.txt = total.txt - delete.txt)

```
import java.io.*;
```

```
class fileExtractor
```

```
{
 public void main(String[] args) throws Exception
```

```
 BufferedReader bai = new BufferedReader(new FileReader
 ("mobile.txt"));
```

```
 PrintWriter pw = new PrintWriter("output.txt");
```

```
 String line = bai.readLine();
```

```
 while (line != null)
```

```
 {
 boolean available = false;
```

```
 BR bai2 = new BR(new FR("delete.txt"));
```

3<sup>rd</sup>

W.A.P to merge data from two files into a 3<sup>rd</sup> file but merging  
Should be done pair by pair. assume that there is a blank line  
B/w Every 2 pairs?

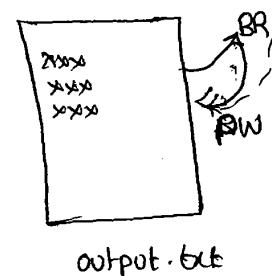
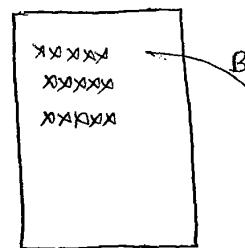
4<sup>th</sup>

W.A.P to delete duplicates from the given i/p file?

```

import java.io.*;
class DuplicateEliminator
{
 public void main(String[] args) throws Exception
 {
 BufferedReader br1 = new BufferedReader(new FileReader("i/p.txt"));
 PrintWriter pw = new PrintWriter("o/p.txt");
 String line = br1.readLine();
 while(line != null)
 {
 boolean available = false;
 BufferedReader br2 = new BR(new FR("o/p.txt"));
 String target = br2.readLine();
 while(target != null)
 {
 if(line.equals(target))
 {
 available = true;
 break;
 }
 target = br2.readLine();
 }
 if(available)
 pw.println(line);
 line = br1.readLine();
 }
 }
}

```



String target = bai.readLine();

while (target != null)

{

if (line.equals(target))

{

available = true;

bBreak;

}

target = bai.readLine();

}

if (available == false)

{

Pw.println(line);

}

line = bai.readLine();

}

Pw.flush();

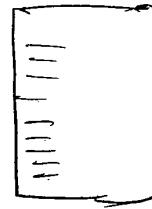
bai.close();

bai.close();

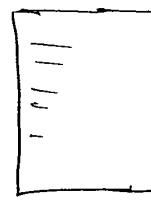
Pw.close();

}

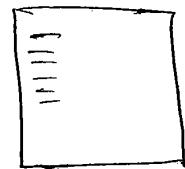
duongjobs.com



total.txt



delete.txt



result.txt

Profession etc

heretic = a person who holds Controversial beliefs, especially Contrary to religion,  
prolonged argument. opposite

324

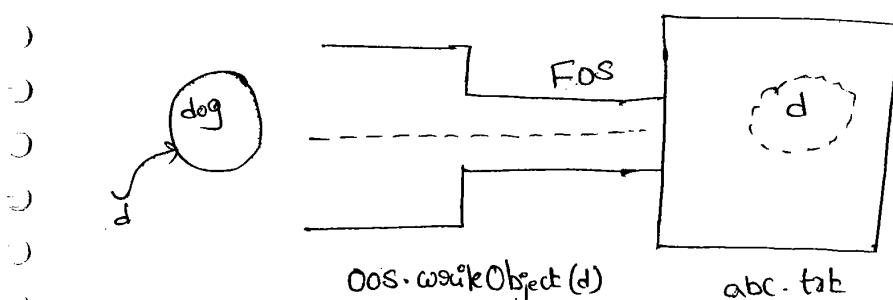
✓ A.

## Serialization

- ① Introduction
- ② Object Graphs in Serialization
- ③ Customized Serialization
- ④ Serialization w.r.t Inheritance.

### Serialization :-

- The process of writing state of an object to a file is called Serialization.
- But strictly it is a process of Converting an Object from java Supported form to either file Supported form or Network Supported form.
- By using `FileOutputStream` and `ObjectOutputStream` classes we can achieve Serialization.

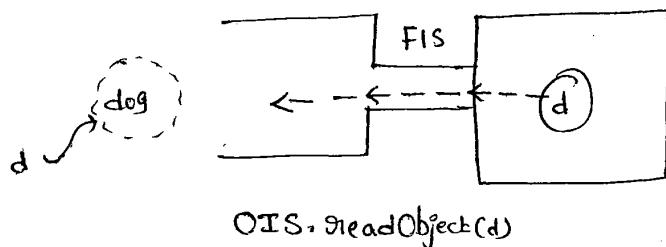


### Deserialization:-

- The process of reading state of an object from a file is called Deserialization.

→ But Strictly Speaking, It is the process of Converting an Object from either Network Supported form or fileSupported form to java Supported form.

→ By using **FileInputStream** and **ObjectInputStream** classes we can achieve De-Serialization.



Ex:-

```
import java.io.*;
Class Dog implements Serializable
{
 int i = 10;
 int j = 20;
}
Class SerializableDemo
{
 p.s.v.m() throws Exception
}
Dog d1 = new Dog();
```

Serialization |  
FileOutputStream fos = new FileOutputStream("abc.txt");  
ObjectOutputStream oos = new ObjectOutputStream(fos);  
oos.writeObject(d1);

FileInputStream fis = new FileInputStream("abc.txt");

ObjectInputStream ois = new ObjectInputStream(fis);

Dog d2 = (Dog) ois.readObject();

System.out.println(d2.i + " --- " + d2.j);

}

→ We can perform serialization only for Serializable objects.

→ An object is said to be Serializable iff the corresponding class

implements Serializable interface.

→ Serializable interface present in java.io package

and doesn't contain any methods, it is a marker interface.

→ If we are trying to serialize a nonSerializable object we will

get Run-time exception saying NotSerializableException.

Transient Keyword :-

→ At the time of serialization if we don't want to serialize the

value of a particular variable to meet the security constraint

we have to declare those variables with "transient" keyword.

→ At the time of serialization Jvm ignores original value of

transient variable and saves default value.

### transient Vs Static :-

→ Static variables are not part of object hence they won't participate in serialization process. Due to this declaring a static variable as transient there is no impact.

### transient Vs final :-

→ final variables will be participated into serialization directly by their values hence declaring a final variable with transient there is no impact.

### Summary :-

| declaration                                           | o/p         |
|-------------------------------------------------------|-------------|
| ① int i=10<br>int j=20                                | 10 ----- 20 |
| ② transient int i=10;<br>int j=20                     | 0 ----- 20  |
| ③ transient final int i=10;<br>transient int j=20;    | 10 ----- 0  |
| ④ transient int i=10;<br>transient & static int j=20; | 0 ----- 20  |

## Object Graph in Serialization :-

→ whenever we are trying to Serialize an object the set of all objects which are reachable from that object will be Serialized automatically this group of objects is called "Object Graph".

→ In Object Graph every object should be Serializable otherwise we will get 'NotSerializableException'.

Ex:-

```

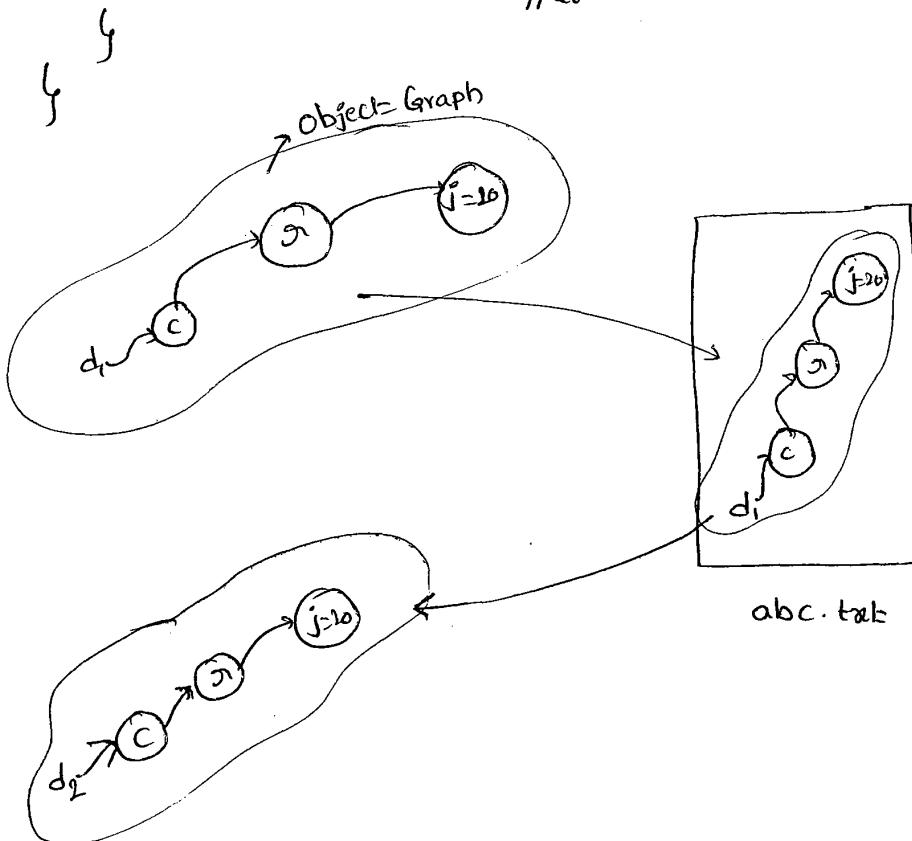
import java.io.*;
class Dog implements Serializable
{
 Cat c = new Cat();
}
class Cat implements Serializable
{
 Rat r = new Rat();
}
class Rat implements Serializable
{
 int j = 20;
}
class SerializeDemo2
{
 p.s.v.m(String[] args) throws Exception
 {
 Dog d = new Dog();
 FileOutputStream fos = new FileOutputStream("abc.ser");
 ObjectOutputStream oos = new ObjectOutputStream(fos);
 oos.writeObject(d);
 }
}
```

```
FileInputStream fis = new FileInputStream ("abc.ser");
```

```
ObjectInputStream ois = new ObjectInputStream (fis);
```

```
Dog di = (Dog) ois.readObject();
```

```
s.o.println(di.c.g.i); // so
```



- In the above prog. whenever we are Serializing a Dog object automatically Cat & Rat objects will be Serialized because these are the part of object graph of dog.
- Among dog, Cat & Rat if atleast one class is not Serializable then we will get NotSerializableException.

## Customized Serialization :-

→ In the default Serialization there may be a chance of loss of information because of transient keyword.

Ex:- class Account implements Serializable

{

String username = "durga";

transient String password = "anushka";

}

Class SerializeDemo2

{

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

{

Account a1 = new Account();

S.O.P(a1.username + "----" + a1.password); durga --- anushka

FileOutputStream fos = new FileOutputStream("abc.ser")

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(a1);

FileInputStream fis = new FileInputStream("abc.ser");

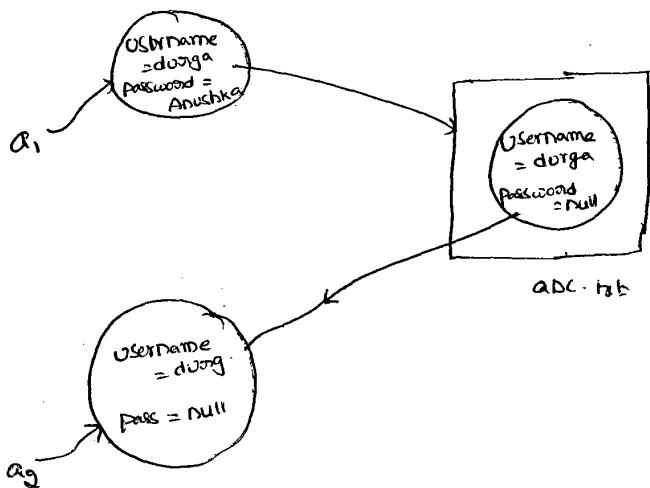
ObjectInputStream ois = new ObjectInputStream(fis);

Account a2 = (Account)readObject();

S.O.P(a2.username + "----" + a2.password);

}

}



→ In the above Example before Serialization Account Object Can provide proper username & password But after deSerialization Account Object Can't provide the Original password. Hence during default Serialization There may be a chance of loss of information due to transient key word. We Can Recover this loss of information by using Customized Serialization.

→ We Can implement Customized Serialization by using the following 2 methods.

(1) private void writeObject(OOS os) throws Exception

→ This method will be Executed automatically at the time of Serialization  
It is a Call back method.

(2) private void readObject(OIS is) throws Exception

→ This method will be Executed automatically at the time of deserialization  
It is a callback method.

→ The above 2 methods we have to define in the Corresponding class of Serialized Object.

Ex:-

```

import java.io.*;
class Account implements Serializable
{
 String username = "dusga",
 transient String pwd = "anushka",
 private void writeObject(ObjectOutputStream os) throws Exception
 {
 os.defaultWriteObject();
 String epwd = (String)os.readObject();
 pwd = epwd.substring(3);
 }
}

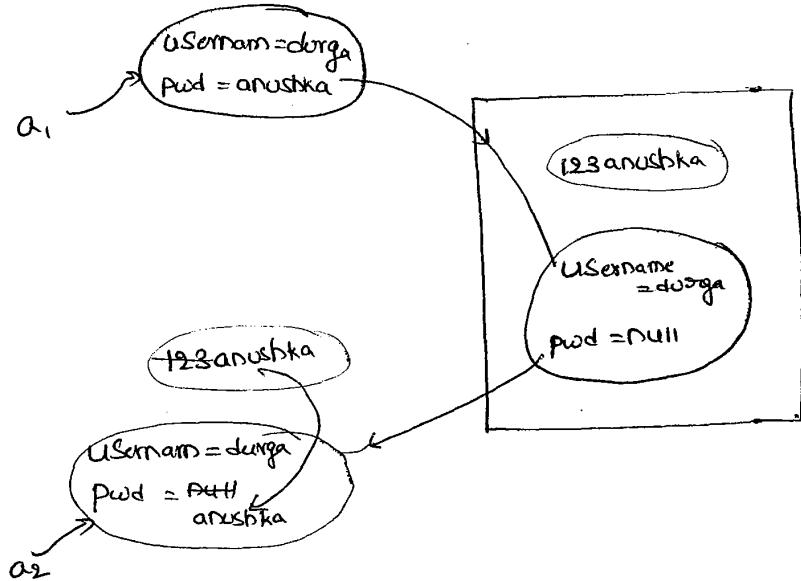
class CashSerializeDemo
{
 public static void main(String[] args) throws Exception
 {
 Account a1 = new Account();
 System.out.println(a1.username + " ---- " + a1.pwd);
 FileOutputStream fos = new FileOutputStream("abc.ser");
 ObjectOutputStream oos = new ObjectOutputStream(fos);
 oos.writeObject(a1);
 FileInputStream fis = new FileInputStream("abc.ser");
 ObjectInputStream ois = new ObjectInputStream(fis);
 Account a2 = (Account)ois.readObject();
 }
}

```

```
S.O.println(a2.getUsername() + " --- " + a2.getPassword());
```

```
}
```

Serialization w.r.t Inheritance :-



Serialization w.r.t Inheritance :-

Case1:-

→ If the parent class implements `Serializable` then every child class is by default `Serializable`. i.e., `Serializable` nature is inheriting from parent to child (`P + C`).

Ex:- Class `Animal` implements `Serializable`

```
d
int x=10;
f
class Dog extends Animal
d
int y=20;
```

→ we can serialize dog object even though dog class doesn't implement Serializable interface explicitly because its parent class Animal is Serializable.

Case 2:-

→ Even though parent class doesn't implement Serializable & if the child is Serializable then we can serialize child class object. At the time of serialization JVM ignores the original values of instance variables which are coming from non-Serializable parent & store default values.

→ At the time of deserialization JVM checks is any parent class is Non-Serializable or not, JVM creates a separate object for every Non-Serializable parent & share its instance variables to the current object.

→ for this JVM always calls no argument constructor of the non-Serializable parent. If the non-Serializable parent doesn't have no argument constructor then we will get RuntimeException.

Ex:-

```
import java.io.*;
```

```
class Animal
```

```
{
```

```
int i=10;
```

```
-Animal()
```

```
{
```

```
S.o.println("Animal constructor called");
```

```
}
```

```
}
```

Class Dog extends Animal implements Serializable

↓

int j = 20;

Dog()

d

S.o.println("Dog Constructor Called");

g

Class SerializeDemo6

↓

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

↓

Dog d = new Dog();

d.i = 888;

d.j = 999;

fileOutputStream fos = new FileOutputStream("abc.ser");

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeObject(d);

S.o.println("Deserialization Started");

fileInputStream fis = new FileInputStream("abc.ser");

ObjectInputStream ois = new ObjectInputStream(fis);

Dog d1 = (Dog)ois.readObject();

S.o.println(d1.i + " " + d1.j);

↓

↓

Op! - Animal Constructor Called

331

Dog " "

Deserialization Started

\* Animal Constructor Called

10 ~~~ 999

