Durgasoft SCJP Notes

Part-1

21/02/11

Language fundamentals

1. Identifiers 2
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 arguements 33
10. Java Coding Standards 34

MANOJ PATEL RIGHTS & ZERO
PLOT NO. 8 SAYAPURI NAGAR
AMEERPET HYDERABAD

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        → classname
{
    p.s.v. main(String [] args)   → method name
    {
        int x=10;
    }
}
```

x → variable name

✓ → is identifiers.

* Rules to define identifiers :-

1) The only allowed characters in Java identifier are :

✓ (a to z, A to Z, 0 to 9, _, $)

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

Ex:-

✓ all_member

✗ all#

✓ _$_$

✗ 098$_10

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

✓ total123.

3). Java identifiers are CaseSensitive.

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

We Can differenciate w.r.t Case.

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

5) <u>Reserved words</u> Can't be used as identifiers.

6) All predefined Java class names & interface names we Can use as identifiers. ~~but~~ Eventhough it is legal, but it is <u>not recommended</u>.

Ex:-

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

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

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

✓ ① Java2share
✗ ② 4shared
✗ ③ all@hands
✓ ④ total_no_of_Students
✓ ⑤ _$_
✗ ⑥ total#
✗ ⑦ int
✓ ⑧ Integer

Primitive data-types (8)

- **Numeric datatypes** (To represent numbers)
  - **Integral datatypes** (to represent whole no.s)
    - Byte
    - short
    - int
    - long
  - **Floating-point datatypes** (to represent real numbers)
    - float
    - double.
- **Char datatypes** (to represent characters)
- **Boolean data types** (to represent logical values)

12.7

① Byte :-

Size = 8-bits (or) 1 Byte

Max_value = 127

Min_value = -128

Range = -128 to +127

MSB

0	1	1	1	1	1	1	1

Sign bit

if → 0 → 127
if 1 → -128

0 → +ve
1 → -ve

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

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

Ex:-

✓ byte b = 100;

✓ byte b = 127;

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

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

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

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

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

② Short :-

Size : 2-bytes (16-bits)

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

[-32768 to 32767]

Eg:- ✓ Short s = 32767

✓ Short s = -32768

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

✗ Short s = 123.456 C.E:- PLP

found : double

Required : Short

✗ 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 completly 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 to 2147483647)

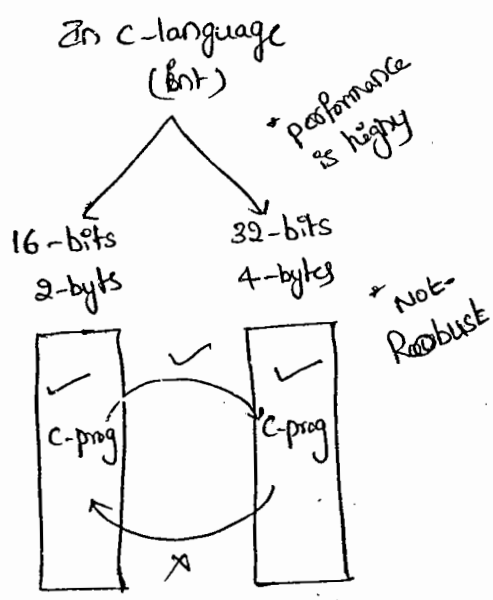
Note :-

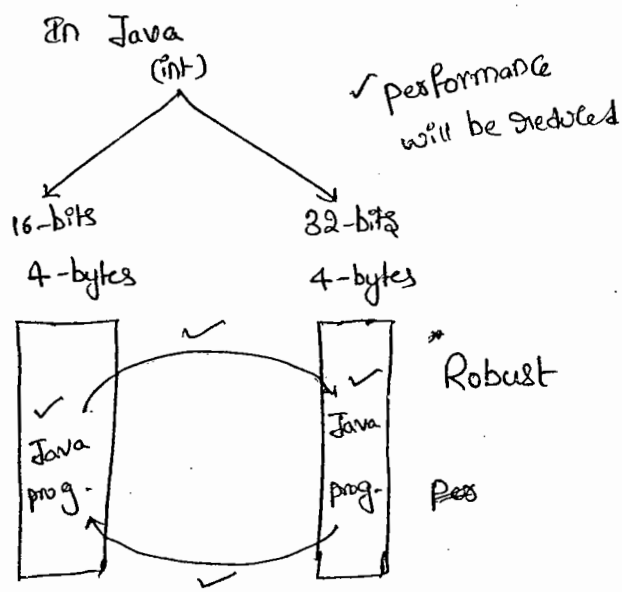
→ In C language the size of int is varied from platform to platform for 16-bit processors it is 2-bytes but for 32-bit processors it is 4-bytes

* The main advantage of this approach is read & write operation we can perform very effiently and performance will be improved. But the main
* disadvantage of this approach is the chance of failing C program is very very high if we are changing platform. Hence C-language is not considered as Robust.

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

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





13/02/11

4) long :-

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

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

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

2)

Ex(2):-

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

Size = 8 bytes

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

Note:-

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

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

Floating Point data types :-

floating point data-types

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

Boolean data type :-

Size : Not Applicable (virtual machine dependent)

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

Q) Whic of the following boolean declarations are valid

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

✓ 2) boolean b = true;

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

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

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

Exp:-

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

in Java ✗

C++ ✓

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

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

in Java ✗

in C++ ✓

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

Char datatype :-

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

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

Size : 2-bytes

Range : 0 to 65535

Summary of primitive data types :-

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

Literals :-

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

Ex:- int x = 10 ;

- int → datatype / Keyword
- x → Name of variable | identifier
- 10 → Constant value | Literal.

Integral Literals :-

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

1) decimal literals :-

allowed digits are 0 to 9

Ex:- int x = 10;

2) Octal literals :-

→ allowed digits are 0 to 7

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

Ex:- int x = 010;

3) Hexadecimal literals :-

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

→ for the Extra digits we Can use both upper case & lower case.

This is one of very few places where Java is not Case sensitive

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

Ex:- int x = 0x10

(or)

int x = 0X10

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

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

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

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

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

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

Quiz

Q) which of the following declarations are valid.

✓ ① int x = 10;

✓ ② int x = 066;

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

✓ ④ int x = 0XFACE; 64206

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

✓ ⑥ int x = 0XBea; 3050

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

Ex:-

✓ 1) int i = 10;

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

✓ 3) long l = 10l;

✓ 4) long l = 10;

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

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

Ex:- byte b = 10; ✓

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

Floating Point Literals:-

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

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

Ex:- ✗ 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:- PLP

found : double
required : float

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

Ex:-

✓ 1) double d = 123.456;

✓ 2) double d = 0123.456; O/P:- 123.456

✗ 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; // 64206.0

✓ 5) float f = 0xBea;

✓ 6) float f = 0642; // 418.0

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

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

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

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

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

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

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

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

✓ 3) float f = 1.2e3f;
o/p:- 1200.0

Boolean Literals:-

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

Q) which of the following Boolean declarations are valid.

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

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

✓ ③ boolen b = true;

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

1) Ex:- int x = 0;

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

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

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

Ex②:-

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

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

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

O/P: Hi

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

O/P:- Hi

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

O/P:- Hello.

Char Literals :-

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

Ex:- ✓ char ch = 'a';

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

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

వివరణ:

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

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

Ex:- ✓ 1) char ch = 97;
S.o.p(ch); a

✓ 2) char ch = 65535;
S.o.pln(ch);

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

✓ 4) char ch = 0XFACE;

✓ 5) char ch = 0642;

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

Ex:- ✓ 1) char ch = '\u0061';

S.o.p(ch); a

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

✓ 3) char ch = '\uface';

✗ 4) char ch = '\i beaf';

4. → Every escape character is a char literal

Ex:- ✓ 1) char ch = '\n';

✓ 2) char ch = '\t';

✗ 3) char ch = '\l';

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

Q) which of the following are valid char declarations.

✓ 1) char ch = 0xbeaf;

✗ 2) char ch = \u beaf; because ' '

✗ 3) char ch = -10;

✗ 4) char ch = '\x';

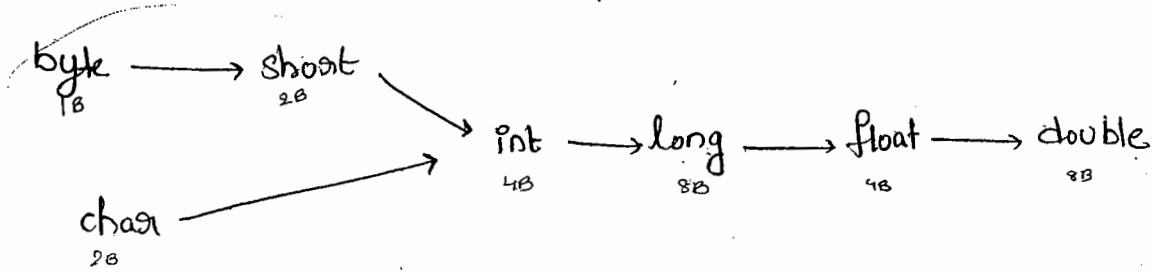
✓ 5) char ch = 'a';

String Literals :-

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

Ex:- String s = " java";

→ The following promotions will be performed automatically by the Compiler



Arrays
(20)

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

Array:-

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

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

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

→ we Can resolve this problem by using Collections.

1) Array declarations:-

(a) Single dimenshional Array declaration:-

✓ 1) int[] a;

✓ 2) int a[];

✓ 3) int []a;

→ 1st one recommanded because Type is clearly seperated from the Name.

→ At the time of declaration we can't specify the size.

Ex:- ✗ 1) int[6] a;

(b) 2D Array declaration:-

✓ 1) int[][] a;

✓ 2) int [][]a;

✓ 3) int a[][];

✓ 4) int[] a[];

✓ 5) int[] []a;

✓ 6) int []a[];

c) 3D - Array declarations:-

1) int[][][] a;
2) int a[][][];
3) int [][][]a;
4) int[] [][]a;
5) int[] a[][];
6) int[] []a[];
7) int[][] []a;
8) int [][] a[];
9) int [][]a[];
10) int []a[][];

Q) Which of the following are valid declarations.

✓ 1) int[] a,b; a→1, b→1

✓ 2) int[] a[],b; a→2, b→1

✓ 3) int[] []a,b; a→2, b→2

✓ 4) int[] []a,b[]; a→2, b→3

✗ 5) int[] []a,[]b; a→2, b→3 C.E:-

→ If we want to Specify the dimenshion before the variable it is possible only for the first variable.

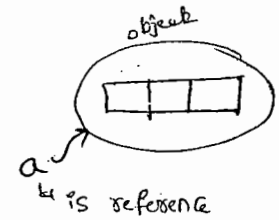
Ex:- int[] []a, []b;

Allowed — not allowed

9) Array Construction :-

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

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



→ for every array type corresponding classes are available. but these classes are not applicable for programmer level.

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

→ At the time of construction compulsary we should specify the size otherwise we will get C.E.

ex:- int[] a = new int[]; ✗ C.E:-

int[] a = new int[3]; ✓

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

ex:- int[] a = new int[0]; ✓

→ If we are specifying array size as -ve int value, we will get runtime exception saying NegativeArraySizeException.

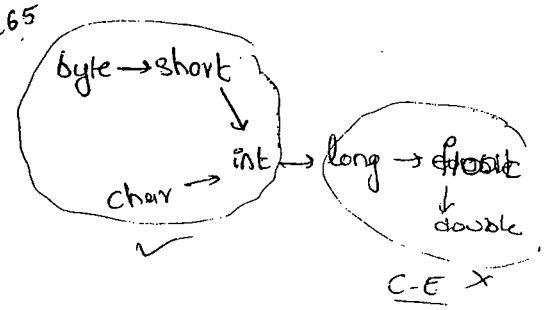
ex:- ✗ int[] a = new int[-6]; R.E:- NegativeArraySizeException

→ To Specify array size the allowed 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)

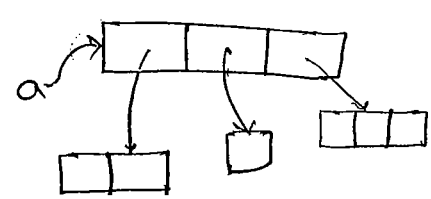
Creation of 2D-Arrays:-

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

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

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

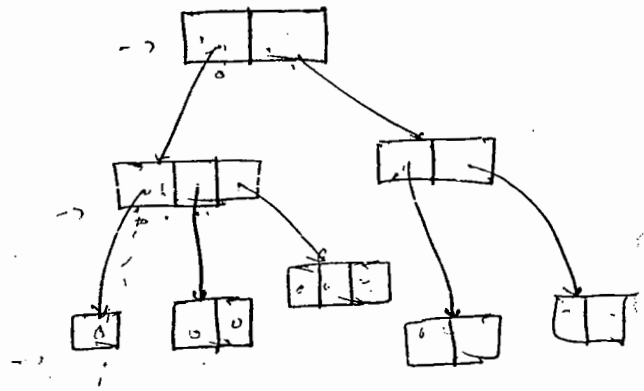
a[0] = new int[2];
a[1] = new int[1];
a[2] = new int[3];



Note:-
In C++, as

Ex 2:

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



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

✗ ① int[] a = new int[];

✓ ② int[][] a = new int[3][2];

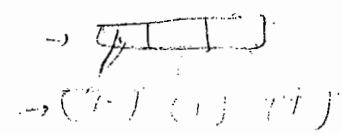
✓ ③ int[][] a = new int[3][];

✗ ④ int[][] a = new int[][2];

✓ ⑤ int[][][] a = new int[3][4][5];

✓ ⑥ int[][][] a = new int[3][4][];

✗ ⑦ int[][][] a = new int[3][][5];



int[][] arr = {{}};

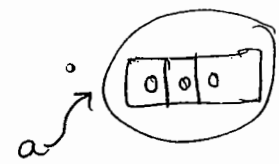
Array Initialization:-

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

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

S.o.pln(a); [I@3e25a5 ↳ hashcode

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

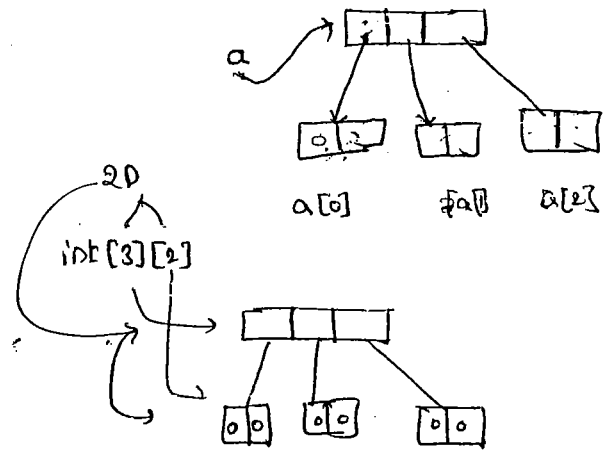


Note:- Whenever we are trying to print any object reference internally toString() will be call which is implemented as follows.

Classname @ hexadecimal_String_of_hashCode.

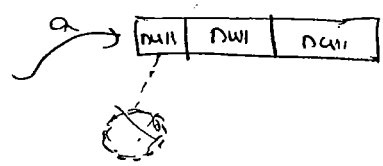
Ex (2):-

```
int[][] a = new int[3][2];
S.o.pln(a);      [[I@-----
S.o.pln(a[0]);   [I@4567
S.o.pln(a[0][0]); 0
```



Ex (3):-

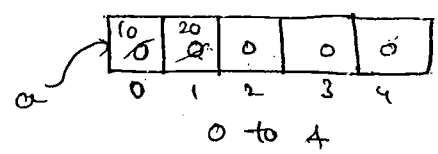
```
int[][] a = new int[3][];
S.o.pln(a);      [[I@-----
S.o.pln(a[0]);   null
S.o.pln(a[0][0]); R.E:- NPE
```



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

Ex:-

```
int[] a = new int[5];
a[0] = 10;
a[1] = 20;
a[3] = 40;
a[50] = 50;   → R.E: AIOBE
a[-50] = 60;  → R.E: AIOBE
a[10.5] = 30; → C.E:- PLP, found = double, required = int.
```



Note:-

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

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

→ We can declare, Construct & Initialize an array into a Single line.

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

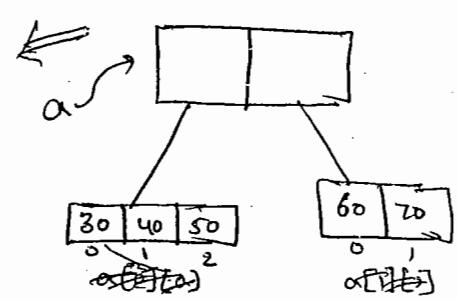
Ex(2):- `char[] ch = {'a', 'e', 'i', 'o', 'u'};`

`String[] s = {"Sunu", "Ravi", "Laxmi", "Sundar"};`

→ We can extend this shortcut even for multidimensional arrays also.

Ex(3):-

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

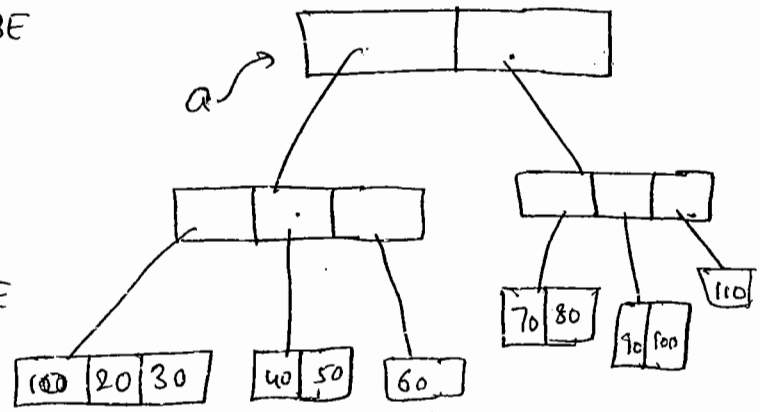


→ We can Extend this Short Cut Even for 3D array also

Ex:-

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

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



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

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

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

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

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

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

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

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

→ If we want to use Shortcut Compulsary we should perform declaration, Construction & initialization in a 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;	✓ int[] x;
✓ x=10	X x = {10, 20, 30};
	C.E:- Illegal Start of Expression.

length() vs length :-

length :-

→ It is a final variable applicable only for arrays.

→ It represents the size of array

Eg:- int[] a = new int[10];

S.o.pln(a.length); 10

S.o.pln(a.length()); C.E

Cannot find Symbol

Symbol : method length

location : class int[]

length() :-

→ It is a final method applicable only for String objects

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

eg:- String s = "durga";

S.o.pln(s.length()); 5

S.o.pln(s.length);

↳ C.E:- Cannot find Symbol

Symbol : variable length

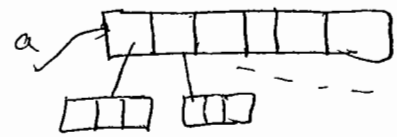
location : java.lang.String.

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

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

S.o.pln(a.length); 6

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



Note:-

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

Annonymous Array :-

→ Sometimes we can create an array with out name also such type of nameless arrays are called "Annonymus arrays".

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

→ We can create annonymous array as follows.

$\boxed{\text{new int[]\{10, 20, 30, 40\}}}$ ✓

→ At the time of annonymous array creation we can't specify the size, otherwise we will get compiletime error.

Ex:- ✗ new int[4]{10, 20, 30, 40}.

Eg:-

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

```
Sum(new int[]{10,20,30,40});
}

public static void ~~no~~ Sum(int[] x)
{
  int total = 0;
  for (int x1 : x)
  {
    total = total + x1;
  }
  S.o.pln("the Sum : " + total);   100
}
```

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

Eg:-

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

Array element assignments :-

Case(1) :-

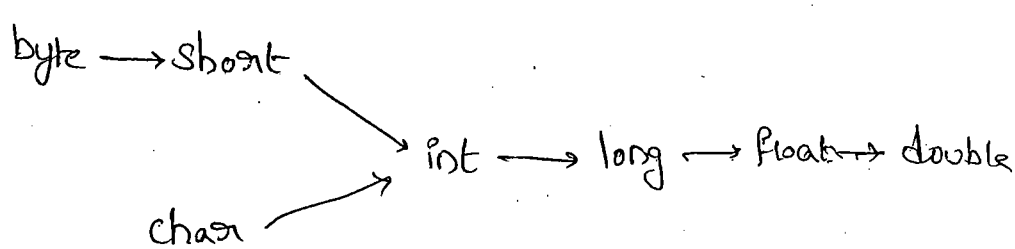
→ for the primitive type arrays as Array elements we can provide any type which can be promoted to declare type.

Eg:- → for the int type arrays, the allowed Element types are byte, short, char, int. if we are providing any other type, we will get Compiletime Error.

Eg(1) :-

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

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



Case(2):-

→ In the case of object type arrays as array elements we can provide either declared type or its child class objects.

eg1:-

① Number[] n = new Number[10];

✓ n[0] = new Integer(10);

✓ n[1] = new Double(10.5);

✗ n[2] = new String("durga"); → C.E:- Incompatable types
found : String
required : Number.

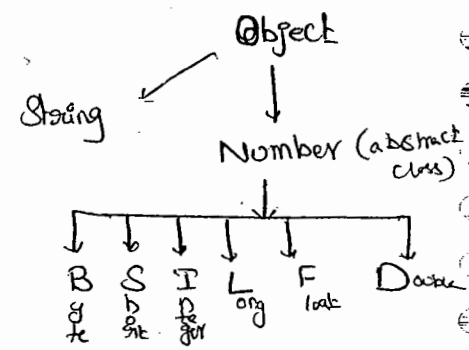
② Object[] a = new Object[10];

✓ a[0] = new Object();

✓ a[1] = new Integer(10);

✓ a[2] = new Double(10.5);

✓ a[3] = new String("durga");



Case(3):-

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

Eg:- ① Number[] n = new Number[10];

✓ n[0] = new Integer(10);

✗ n[1] = new Number();

Case 4 :-

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

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

r[0] = new Thread();

✗ r[1] = new String("durga"); C.E:- Incompatable type
found : String
Required : Runnable

Runnable (I)
↑
Thread (c)

Note :-

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

Array Variable Assignment :-

Case(1):

→ Element level promotions are not applicable at array level

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

① int[] a = {10, 20, 30, 40};

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

✓ int[] b = a;

✗ int[] c = ch; C.E:- Incompatiable type
found : char[]
required : int[]

Q) Which of the following promotions are valid.

✓ ① char → int

✗ ② char[] → int[]

✓ ③ int → long

✗ ④ int[] → long[]

✗ ⑤ long → int

✗ ⑥ long[] → double[]

✓ ⑦ String (child) → Object (parent)

✓ ⑧ String[] → Object[]

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

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

Eg(1) String[] s = {"A", "B", "C"};

✓ Object() a = s;

Case(2):-

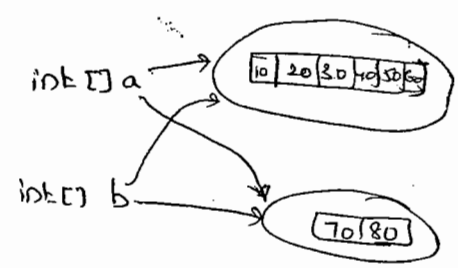
→ When even we are assigning one array to another array only reference variables will be reassign but not underlying elements. Hence types must be matched but not sized.

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

int[] b = {70, 80};

✓ ① a = b;

✓ ② b = a;



Eg(2):

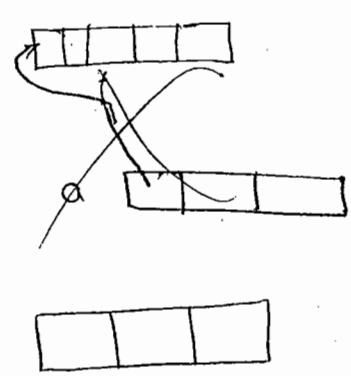
int[][] a = new int[3][2];

a[0] = new int[5];

a[1] = new int[4];

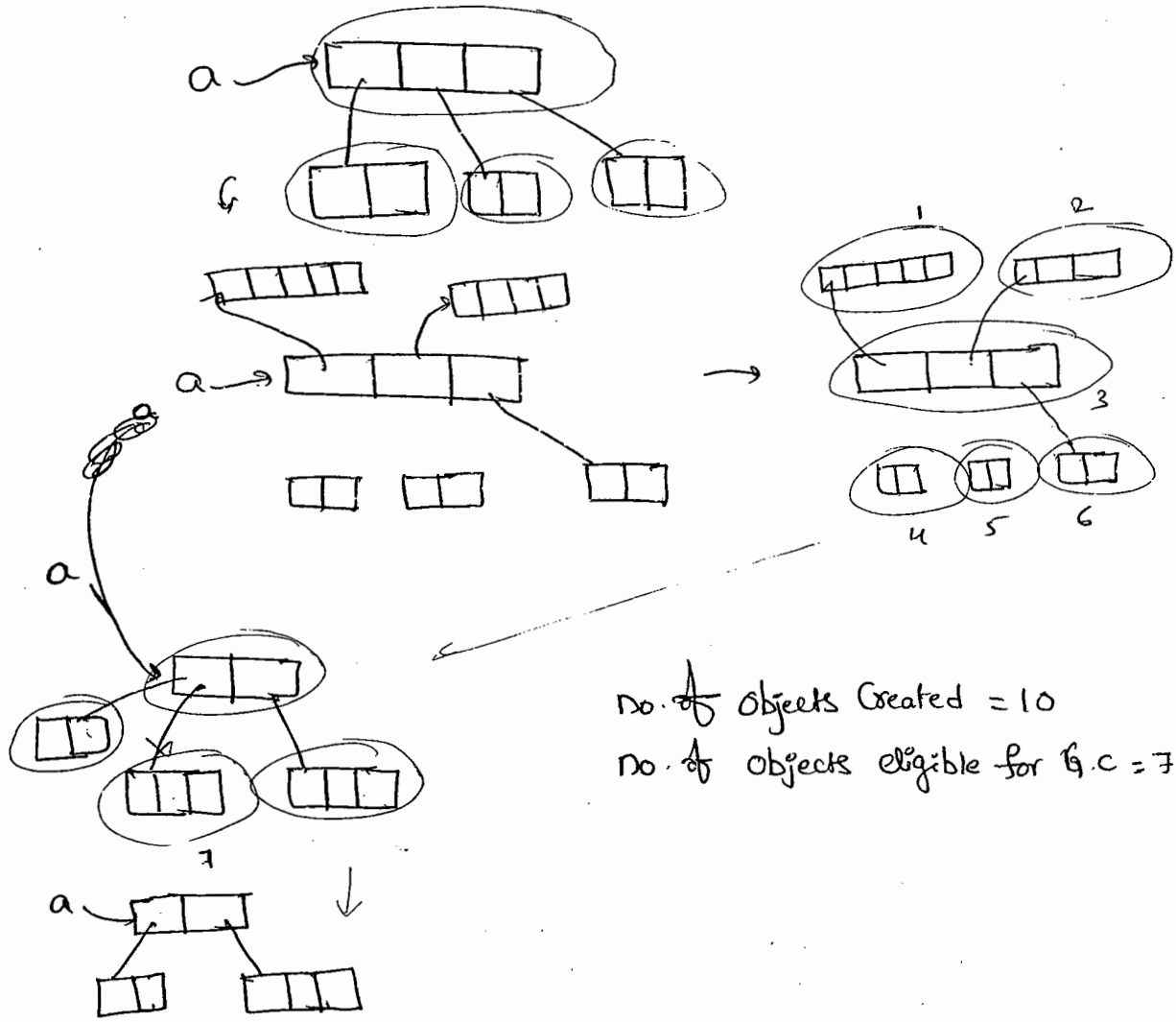
a = new int[2][3];

a[0] = new int[2];



No. of objects Created = 10

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



no. of objects Created = 10

no. of objects eligible for G.C = 7

Case 3:-

→ when ever 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

eg:-

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

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

Types of Variables

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

(i) primitive variables
(ii) reference variables

(i) Primitive Variables:-

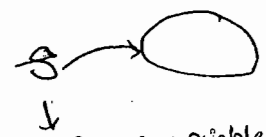
→ Can be used to represent primitive values

ex:- int x = 10;

(ii) reference variables:-

→ Can be used to refer Objects

ex:- Student s = new Student();



s is reference variable

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

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

(i) instance variable :-

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

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

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

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

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

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

→ But from instance area we Can access instance members directly

Ex!.

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

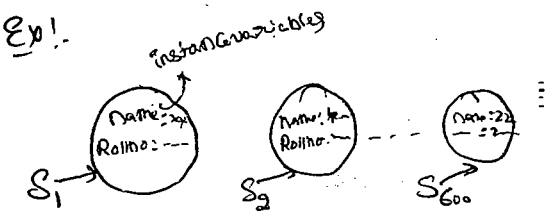
```
        Test t = new Test();
        S.o.pln(t.x); 10
    }
    public void m1()
    {
        S.o.pln(x); 10
    }
}
```

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

eg:-

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

Ex:-



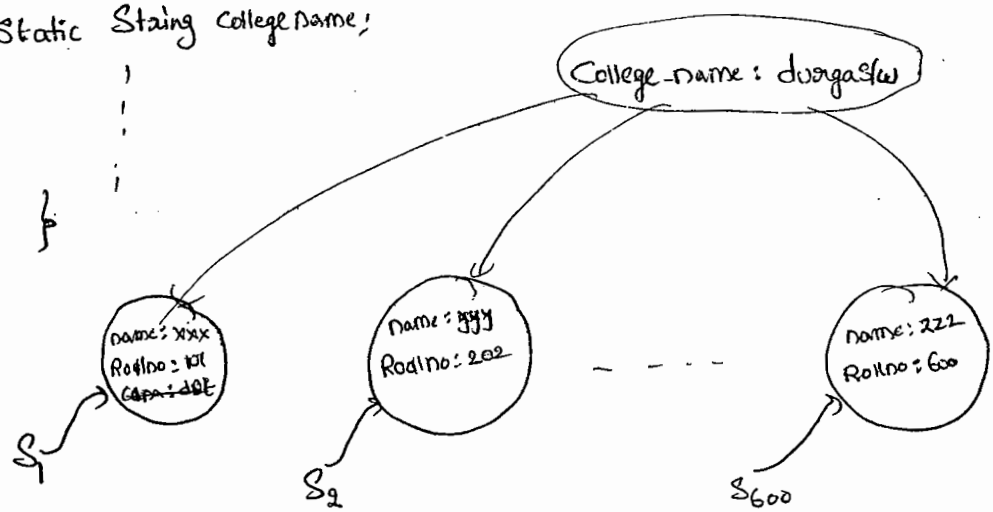
Students objects, in that name, Rollnos are instance variables, Bcz, these values are varied from object to object.

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

(ii) Static variables :-

Ex:-

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



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

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

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

Exactly Same as the Scope of the class.

Note:- Java Test ↵ execution process is

① Start jvm

② Create main Thread

③ Locate Test.class

④ Load Test.class → Static variables Creation

⑤ Execute main() method of Test.class

⑥ unload Test.class → Static variables destruction

⑦ Destroy main Thread

⑧ ShutDown Jvm

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

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

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

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

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

→ Eg:

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

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

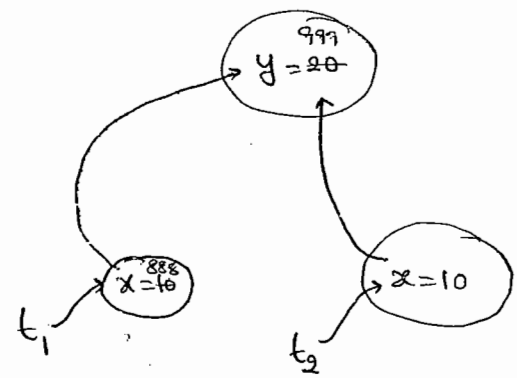
eg:-

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

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

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);
  }
}
```



$t_1.x = 888$ $\quad$ $t_2.x = 10$

$t_2.y = 999$ $\quad$ $t_2.y = 999$

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

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

(iii) Local variables :-

→ To meet temparay requirements of the programer some-times 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 temparay variables.

→ Local variables will be stored inside a Stack.

→ The Local variables will be created while executing the block in which we declared it & destoryed once the Block completed. Hence, the Scope of Local variable is 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.pln (i + "----" + j);
  }
}
```

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

→ For the Local variables JVM won't provide any default values, Compulsary we should perform initialization Explicitly, before using That variable.

Eg:- ①

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

O/P:- Hello

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

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

Eg(2):-

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

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

Eg 3:-

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

O/P:- Java Test ↵

20

Java Test x y ↵

10

→ Note:-

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

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

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

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

Eg:-

```
Class Test
{
  p.s.v.m(String[] args)
  {
  ✗ private int x=10;
  ✗ public int x=10;
  ✗ protected int x=10;
  ✗ static int x=10;
  ✓ final int x=10;
  }
}
```

C.E:- Illegal Start of Expression.

UN Initialized Arrays:-

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

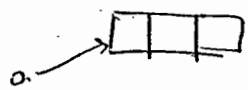
Instance level :-

int[] a ; S.o.p(obj.a) null

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

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

S.o.p (obj.a[0]) 0



Static level :-

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

S.o.p(a[0]); NPE

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

S.o.p(a[0]); 0

Explanation :-

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

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

Local Level :-

int[]a ; S.o.p (a), S.o.p(a[0]) } C.E :- variable a might not have been initialized

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

S.o.p(a[0]) 0

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

8/03/10

Var-arg methods (1.5 version)

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

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

→ We can declare var-arg method as follows.

```
m1(int... x)
```

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

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

Ex(1):-

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

o/p:- var-arg method
var-arg method
" "
" "

→ Internally var-arg method is implemented by using single dimensional arrays concept. Hence with in the var-arg method we can differenciate arrguements by using <u>index</u>.

Ex:-

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

Op:

```
the Sum : 0
the Sum : 30
the Sum : 60
the Sum : 100
```

Case(3):-

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 parameter with normal parameter also.

ex:- m1(int x, String... y) ✓

Case 3:-

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

ex:- m1(int... x, String y) ✗

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.pln("General method");
  }
  p.s.v.m1(int... i)
  {
    S.o.pln("var-arg");
  }
  p.s.v.m(String[] args)
  {
    m1();  var-arg
    m1(10); General (only)
    m1(10, 20); var-arg
  }
}
```

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

Case 6:-

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

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

Var-arg Vs Single dimenshional arrays:-

Case(1):-

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

Ex:- m1(int[] x) ⟹ m1(int... x) ✓

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

Case 2:-

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

✗ m1(int... x) ⟹ m1(int[] x)

09/03/11

main()

main() :-

→ Wheather the Class Contains main() or not & wheather the main() is properly declared or not, these checkings are not responsibilities of Compiler. At runtime, JVM is responsible for these checking.

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

Ex :-

```
class Test
{
}
```

compile Javac Test.java ✓

run ✗ Java Test → R.E :- NoSuchMethodError : main

→ JVM always Searches for the main() with the following Signature.

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

- public → To call by JVM from any where
- static → without existing object also JVM has to call this method
- void → main method can't return anything to JVM
- main → name of method which is Configured inside JVM
- (String[] args) → Command-line arguements.

→ If we are performing any change to the above signature we will get runtime Exception Saying "NoSuchMethodError: main".

→ Any where the following changes are acceptable.

(1) we Can change the order of modifiers. i.e instead of public static we Can take static public.

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

String[] args ✓

String []args ✓

String args[] ✓

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

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

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

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

(i) final

(ii) Synchronized

(iii) strictfp.

Ex:-

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

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

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

(ii) Static public Void Main(String[] args) ✗

(iii) public Synchronized Strictfp final Void main(String[] args) ✗

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

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

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

Ans:- No where, All cases will Compile.

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

Ex:-
```
class P
{
 public static void main(String[] args)
 {
   S.out.p("RLU durga s/w");
 }
}
class C extends P.
{
}
```

javac p.java ✓

java p

O/P :- RLU durga S/w

java C

O/P:- RLU durga S/w

Ex 2:-

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

javac P.java

java P

o/p:- I love

java C

o/p:- durga s/w.

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

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

ex:-

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

o/p:- durga s/w.

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

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

Q) Explain about S.o.pln?

A)

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

Test.name.length()
```

- Test → It is a Class-name
- name → Static variable of type String present in Test class
- length() → it is a method present in String class

```
Class System
{
  Static PrintStream out;
}

System.out.println()
```

- System → It is a Class Name present in java.lang
- out → Static variable of type PrintStream present in System Class
- println() → It is a method present in PrintStream class

10/03/11

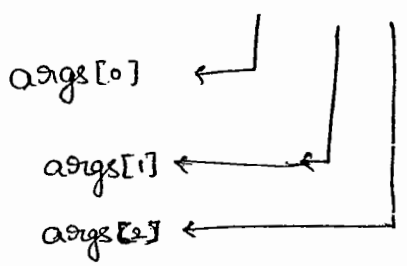
Command Line Arguements

Command line arguements :-

→ The arguements which are passing from Command prompt are called Command Line arguements.

→ The main objective of Command line arguements are we can Customize the behaviour of the main().

Ex:- Java Test x y z



args.length ⇒ 3.

Ex(1):-

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

O/P :- Java Test ↵

R.E :- AIOBE

Java test x y ↵

x

y

R.E :- AIOBE

Ex(2):-

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

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

Java Test 10 20

O/P:- 1020

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

Ex:-
```
class Test
{
  p.s.v.m(String[] args)
  {
    S.o.pln(args[0]); Note Book
  }
}
```

Java Test "Note Book"

Ex(2):-
```
class Test
{
  p.s.v.m(String[] args)
  {
    String[] argh = {"A", "B"};
    args = argh;
    for(String S1 : args)
    {
      S.o.pln(S1);
    }
  }
}
```

```
Java Test x y ↵
o/p A
    B
Java Test x y z ↵
o/p A
    B
Java Test ↵
o/p A
    B
```

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

Java Coding Standards

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

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

~~Ameerpet~~ Standard

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

~~Hi~~tech-City

Coding Standards for Classes:-

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

Ex:- Student
Customer
String
StringBuffer,
} → Nouns

2) Coding Standards for Interfaces :-

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

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

Note :-

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

3) Coding Standards for Methods :-

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

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

getName()
setSalary()
⋮
} Verb + noun

(4) Coding Standards for Variables :-

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

Ex! name
rollno
mobile Number
⋮
} → nouns

⑧ Coding Standards for Constants:-

→ Usually the Constants are Nouns. Should Contain only Upper Case Characters. If it Contains multiple words, These words are Seperated with "_" symbol.

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

ex!-
MAX_VALUE
MIN_VALUE
MAX_PRIORITY
MIN_PRIORITY

⑨ Java bean Coding Standards

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

ex.-

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

→ ends with Bean is not official Convention from SUN.

Syntax for setter method :-

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

Syntax for getter method :-

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

→ It should be no arrguement method.

→ returntype should not be void.

Note :-

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

ex:-

```
private boolean empty;

public boolean getEmpty()
{
    return empty;
}

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

① Coding Standards for Listeners :-

* To register a listener :-

→ method name should be prefix with add,

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

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

✗ ② public void registerMyActionListener(MyActionListener l)

✗ ③ public void addMyActionListener(Listener l)

To unregister a Listener:-

→ The rule is same as above, Except method name should be prefix with remove.

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

✗ ② public void unregisterMyActionListener(MyActionListener l)

✗ ③ public void deleteMyActionListener(MyActionListener l)

✗ ④ public void removeMyActionListener(ActionListener l)

Note:-

In Javabean Coding Standards & Listener Concept 1 compulsary.

Operators & Assignments

Kathy Sierra 1.6
book for SCJP.

Increment / Decrement 2

Arithmetic operators 3

Concatenation 5

Relational operators 5

Equality operators 6

Bitwise operators 7

Short-Circuit 9

instanceof 6

typeCast Operator 10

Assignment Operator 12

Conditional Operator 13

new operator 13

[] operator 13

Operator precedence 14

Evaluation Order of Java operands. 14

Increment & Decrement Operators:

Increment

- Pre-increment: `int x = ++y;`
- Post-increment: `int x = y++;`

Decrement

- Pre-decrement: `int x = --y;`
- Post-decrement: `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;
int y = ++4;
S.out.p.ln(y);
```

C.E: unExpected type
found : Value ②
required : Variable ①

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

```
int x = 4;
int y = ++(++x);
S.o.p(y);
```

after since - it is - constant then

C.E: unexpected type

② found : value

① Required : variable

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

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

Ex(2):- final int x = 4;
x = 5

C.E:- Can't assign a value to final variable x.

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

① double d = 10.5;
d++;
S.o.p(d); // 11.5

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

③ boolean b = true;
++b;
S.o.pln(b);

C.E:-
operator ++ can't applied to boolean.

④ int x = 10;
x++;
S.o.pln(x); //

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

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

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

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

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

③ ✓
```
byte b=10
b = (byte) (b+1)
S.o.pln(b);// //
```

④
```
byte a =10;
byte b =20;
byte c =a+b;
S.o.pln(c);
```

C.E:- PLP
f = int
R = byte

Explanation:-

Max(int, 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,

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

```
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++ ; ⟹ b = (byte) (b+1);

b++ ; ⟹ b = (type of b) (b+1);

Arithmatic operators:-

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

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

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

byte + byte = int

byte + short = int — S.o.pln (10 + 0.0); // 10.0

int + long = long — S.o.pln ('a' + 'b'); 195

long + float = float — S.o.pln (100 + 'a'); 197.

double + char = double

char + char = int

Infinity:-

→ In the Case of integral arithmatic (int, short, long, byte), there is no way to represent infinity. Hence, if the infinity is the result we will always get Arithmatic Exception. (AE : 1 by zero)

eg:-

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

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

Positive_Infinity = Infinity
Negative_Infinity = -Infinity

+ve-∞ = ∞
-ve-∞ = -∞

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

Eg:- ①. S.o.pln (10/0.0) ; Infinity

②. S.o.pln (-10/0.0) ; -Infinity.

* NaN :- (Not a Number)

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

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

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

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

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

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

Ex: * public static double Sqrt (double d);

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

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

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

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

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

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

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

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

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

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

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

$x > NaN$
$x >= NaN$
$x < NaN$
$x <= NaN$
$x == NaN$
} false

Conclusion about A.E (ArithmeticException) :-

→ It is RuntimeException but not Compile time Error.

→ possible only in Integral Arithematic but not floating point Arithmatic
(int, byte, short, char) (float, double)

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

5. String Concatination operator (+)

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

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

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

String d = "shanth";

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

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

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

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

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

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

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

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

String c = "shanth";

× a = (b+c); → total String C.E:- Incompatible type : found : String
Required : int

✓ c = a+c; (String, total String)

✓ b = a+b; (int, int+int)

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

A=65, a=97

Relational Operators

These are >, <, >=, <=

1) We can apply Relational operators for Every primitive datatype Except boolean.

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

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

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

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

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

3) Nesting of Relational operators 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
s2 → durga

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

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

Equality Operators (==, !=)

→ These are ==, !=

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

*** Eg:-

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

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

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

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

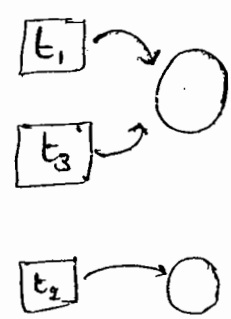
Ex(1): Thread t_1 = new Thread();

Thread t_2 = new Thread();

Thread t_3 = t_1;

✗ S.o.p (t_1 == t_2) ; false

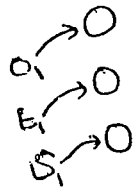
✓ S.o.p (t_1 == t_3) ; True



*→ To apply Equality Operators b/w the object references compulsory there should be some relationship b/w arrguement types.

[either parent to child (or) child to parent (or) same type] otherwise we will get CE: Incomparable type].

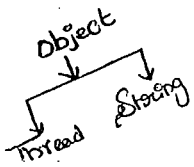
Eg:-(2):- object o_1 = new Object(); because object is Super class



Thread t_1 = new Thread();

String s_1 = new String("shanth");

S.o.p(t_1 == s_1); CE:- Incomparable types Thread & java.lang.String



S.o.p(t_1 == o_1); F

S.o.p(s_1 == o_1); F

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

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

Note:-

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

Instanceof operator (instanceof) ✓

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

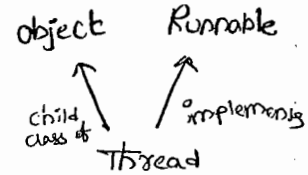
Syn:- $\boxed{r \text{ instanceof } X}$

instanceof
Hashtree
Stricfp

any reference type — class/interface.

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

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



✓ S.o.p(t instanceOf Thread) ; True

✓ S.o.p(t instanceOf Object) ; True

✓ S.o.p(t instanceOf Runnable) ; True

↳ To use instanceOf operator, Compulsary there should be some relationship b/w arrgument type, otherwise we will get Compile-time Error Saying <u>Inconvertable type.</u>

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

S.o.p(t instanceOf String); C.E:-

Inconvertable type

found : Thread

Required : String

↳ whenever we are checking parent object is of child type then we will get <u>false as output</u>.

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

✓ S.o.p(o instanceOf String) ; false

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

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

Eg:- Iterator itr = l.iterator();
while (itr.hasNext())
{
Object o = itr.next();
if (o instanceOf Student)
{ Apply Student related function }
else if (o instanceOf Cu)
{ Apply customer related }

Bit-wise Operators :-

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

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

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

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

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

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

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

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

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

→ We can apply these operators Even for integral data-types also.

also.

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

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

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

Bitwise Complement Operator (~) :- (Filed)

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

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

Ex:- 1) S.o.pln(~True);

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

✓ 2) S.o.pln(~4); -5

4 = 0000 0000 ---- 0100

~4 = 1111 1111 ---- 1011

0 → +ve
1 → -ve

2's Complement

-ve

One's Comp

000 0000 ---- 0100

2's Comp 0 1

000 ---------- 0101

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

-ve 5

∴ -5

Note:

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

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

Boolean Complement Operator (!) :-

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

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

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

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

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

Summary:-

& | ^ ⟹ we can apply for both integral & boolean types.

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

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

Short-Circuit Operators (&&, ||)

→ double AND
→ double OR

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

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

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

Ex:- if (~~~~ & ~~~~)

e_1 → 10 min, & → 1 min, e_2 → 10 min

+ 21 min

{

}

else

{

}

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

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

Ex:-

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

O/P:-

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

(2)

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

Ans:

a) C.E

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

c) Hello

d) Hi

Note:

if we Replace && with &

then Result is (b), that is R.E.

a=97
A=65

TypeCast Operators:-

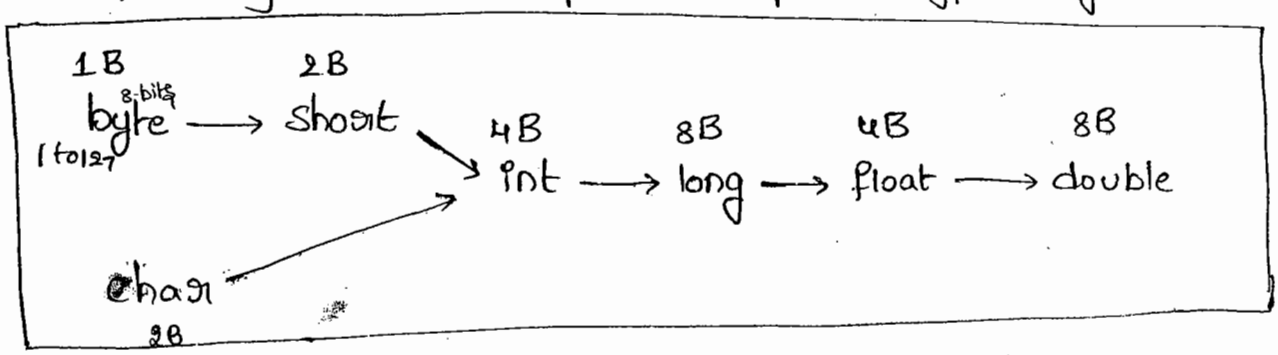
→ there are 2 types of primitive type Castings.

1. Implicit type Casting
2. Explicit type Casting.

Implicit TypeCasting:-

1) Compiler is the responsible to perform this typeCasting
2) this TypeCasting is required when ever we are assigning smaller data type value to the bigger data type variable.
3) It is also known as " widening (or) upCasting ".
4) No loss of information in this type Casting.

→ the following are various possible implicit typeCasting



Ex(1):-

① double d = 10; [Compiler Converts int to double automatically]

/ S.o.pln(d); 10.0

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

/ S.o.pln(x); 97

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

2) Explicit type Casting :-

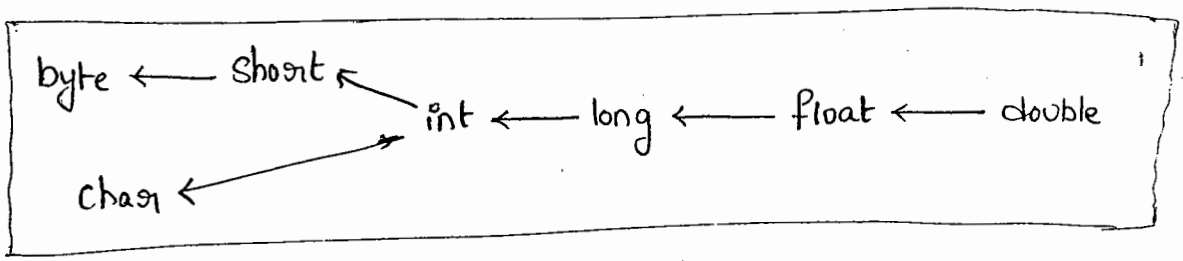
1) programmer is responsible to perform this TypeCasting

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

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

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

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



Ex :-

1) ✗ | byte b = 130

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

2) byte b = (byte) 130 ;

S.o.p(b) ; -126

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

① ✗ byte b = 130 ;

✓ byte b = (byte) 130 ;

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

130 ≡ 0000 - - - - - - - - - 10000010 (32-bits)

byte b ≡ 10000010 (8 bit)

-ve

1's Complement

0000010
1111110

1111101
1
1111110

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

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

126

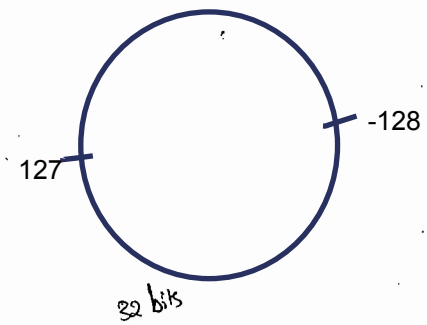
∴ −126

②

int i = 150 ;

short s = (short) i ;

S.o.pln(s) ; 150



150 ≡ 0000 - - - - - 01001011 0

short s ≡ 0000 - - - 01001011 0 → 2 Bytes = short = 16-bits

+ve

don't apply 2's Comp.

∴ s = 150

③ int x = 150 ;

byte b = (byte) x ;

Short s = (short) x ;

S.o.pln(b) ; −106

S.o.pln(x) ; 150

150 ≡ 0000 - - - 01001011 0

byte b = 10010110

-ve

2's Com

1101010

1101001
1
1101010

∴ −106 = 2 + 8 + 32 + 64 = 106

10/2/11

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

Ex:-

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

Assignment Operators :-

→ There are 3 types of assignment operators

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

1. Simple assignment operator :-

Ex:- int x = 10;

2. chained assignment operator :-

Ex:-

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

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

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

C.E: Can't find Symbol

Symbol : variable b

location : Class Test

int a = b = c = d = 20 ;

(Same c & d)

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

3. Compound assignment operator :-

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

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

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

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

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

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

Ex ①

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

C.E: PLP
found : int
required : byte
b = b+1;

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

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

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

Ex ②:-

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

620 600 30 10

Conditional Operator (?:)

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

a+b → binary operator
++a → unary "
(a+b)? a:b ; → ternary "

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

a>b is T then 40
a>b is F then 50

→ Nesting of Conditional operator is possible.

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

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

F F

S.o.pln(x) ; 999

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

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

✓ a<12 T

✗ a<b ✗ C.E

don't Compare these variables

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

C.E:- PLP
found : int
required : byte.

final int a=10, b=20;

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

New operator :-

→ We Can use this operator for creation of objects.

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

[ ] operator :-

→ we Can use these operator for declaring & Creating arrays

Operator precidence :-

1. Unary operators :-

[ ] , x++ , x--

++x , --x , ~ , !

new , <type> (used to type cast)

2. Arithematic Operators :-

* , / , %

+ , -

3. Shift operator :-

>>> , >> , <<

4. Comparision operator :-

< , <= , > , >= , instanceOf

5. Equality operator :-

== , !=

6. Bitwise operators :-

&

^

|

7. Short - Circuit operators :-

&&

||

8. Conditional operators :-

?:

9. Assignment operators :-

= , += , -= , ...

Evalution order of operands :-

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

Ex :-

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

o/p :-

10

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

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

$1 + 6 + 20/6$

$1 + 6 + 3$

$7 + 3$

$= 10$

Ex(2) :-

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

1st increment

2nd place int into x

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

1st place x = 10

∴ x = 10++

↳ x = 11

but last operation is

x = 10

Ex (3) :-

```
int x = 0;
x = ++x + x++ + x++ + ++x;
S.o.p(x) ; 8
```

(1+2)³

x = ~~0~~ ~~1~~ ~~2~~ ~~3~~ 4

x++ = 1

x++ = 2

3

4

Ex 4 :-

```
int x = 0;
x += ++x + x++;
S.o.pln(x) ; 2
```

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

= 0 + 1 + 1

x = 2

Flow Control

16/05/2011 52

Flow Control :-

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

Flow-Control

- **Selection Statements**
  1. if-else
  2. Switch
- **Iterative Statements**
  1. while
  2. do-while
  3. for()
  4. for-each loop
- **Transfer Statements**
  1. break
  2. Continue
  3. return
  4. try
  5. catch
  6. final

(a) Selection Statements :-

(i) if-else :-

Syn :-

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

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

Ex:-

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

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

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

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

o/p:- Hi

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

o/p:- Hello

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

o/p:- Hi

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

Ex:-

```
if(true)
  S.o.p.ln("Hello");
```

✓

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

✗ C.E!

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

✓

```
if(true);
```

✓

Switch Statement :-

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

Syn:-

```
Switch(x)
{
  Case1 :
        Action1;

  Case 2 :
        Action 2;
  :
  default :
        default Action;
}
```

↦ Curlly braces are mandatory.

→ both Case & default are optional inside a Switch

Ex:-

```
int x=10;
Switch(x)
{
}
```

✓

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

Exp:-

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

C.E:-

Case, default or '}' expected

→ until 1.4V the allowed datatypes for switch arguement are

byte
Short
int
char

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

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

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

Ex:-

```
byte b=10;
switch (b)
{
}
```

```
char ch='a';
switch (ch)
{
}
```

```
long l=10l;
switch (l)
{
}
```

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

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

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

→ every case label should be within the range of switch arguement type otherwise we will get compiletime error.

Ex:-

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

(b → byte)

-128 to 127

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

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

(b+1 → int type)

→ every case label should be a valid compile-time constant, if we are taking as variable as case label we will get compile-time error.

Ex:-

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

Suppose final int y=20;

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

C.E:- Constant expression required.

→ If we declare y as final then we wont to get any compile-time error

→ Expressions are allowed for both switch arguement & case label but case label should be constant expression

ex:-

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

→ duplicate Case labels are not allowed.

eg:-

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

C.E:- duplicate Case label

Summary:-

Case label

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

fall-through inside switch :-

→ with in the switch statement if any case is matched from that case onwards all statements will be executed until break statement or end of the switch. This is called fall-through in inside switch.

Ex 1:-

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

  Case 1:
        S.o.pln ("1");
        break;
  Case 2:
        S.o.pln ("2");

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

o/p:-

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

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

Exp:-

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

default case :-

→ We can use default case to define default action.

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

→ we can take default case any where within the switch but it is convension to take as last case.

Exp:-

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

x = 0	x = 1
0	1
	2

x = 2	x = 3
2	def
	0

(b) Iterative Statements :-

(1) While :-

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

Ex:- ① while (rs.next())
{
--
}
Result Set

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

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

Syntax :-

```
while (b)
{
  Action
}
```

boolean type

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

Ex:-

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

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

→ Curlly braces are optional and without Curlly braces we can take only one Statement which should not be declarative statement.

Ex:①

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

```
while(true);
```

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

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

Ex:②:-

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

C.E:- unreachable Statement

②
```
while (false)
{
    S.o.pln("Hello");
}
S.o.pln("Hi");
```

C.E:- unreachable Statement

③
```
int a=10, b=20;
while (a<b)
{
    S.o.pln("Hello");
}
S.o.pln("Hi");
```

O/P:- Hello
Hello
Hello
⋮

④
```
final int a=10, b=20;
while (a<b)
{
    S.o.pln("Hello");
}
S.o.pln("Hi");
```

True

Unreachable Statement

③ do-while :-

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

Syn:-

```
do
{
Action
} while (b);
```

→ should be boolean type

→ mandatory

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

Ex:- ①

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

✓

②

```
do ;
while(true);
```

is a valid java statement

✓

③

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

✗

④

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

✓

⑤

```
do
while(true);
```

→ Compulsary one statement declare (or) take ";"

✗ C.E:-

⑥

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

✓

(or)

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

O/P:- Hello
Hello
⋮

note:-
a ";" is a valid java statement

Eg:- ①

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

C.E: unreachable Statement

②

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

o/p:- Hello
Hi

③

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

o/p:- Hello
Hello
⋮

④

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

o/p:- Hello
Hi

⑤

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

C.E:- unreachable Statement

⑥

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

o/p:- Hello
Hi

for() :-

→ This is the most Commonly used loop

Syntax :-

only once

① ② ⑤ ⑧ ④ ⑦

```
for ( initialisation - Section ; Conditional - Expression ; increment/decrement)
{
    Body ③ ⑥ ⑨
}
```

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

(a) initialization-Section :-

→ This will be executed only once.

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

→ Here we Can declare multiple variables of the Same type but different datatype variables we Can't declare.

Ex:- ① int i=0, j=0; ✓

② int i=0, byte b=0; ✗

③ int i=0, int j=0; ✗

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

Ex!- int i=0;

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

O/P!- Hello UR Sleeping

No Boss U only Sleeping

No Boss U only Sleeping

No Boss U only Sleeping

Conditional Expression:-

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

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

Increment & decrement Section :-

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

Ex!- int i=0;

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

O/P!- Hello
Hi
Hi

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

→ All 3 parts of for loop are optional

Ex:- for (; T ;) ; → Statement So, it is True.

⇒ represent infinite loop

Note:-

; is a valid Java Statement

Ex:-

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

C.E:- unreachable

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

C.E:- unreachable

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

C.E: unreachable

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

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

O/P:- Hello
Hello
⋮

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

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

a<b → True

O/P:- C.E:- unreachable statement.

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

→ Introduced in 1.5V. This

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

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

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

for-loop

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

10
20
30
40
50

for-each

```
for (int x : a)
{
  S.o.pln(x);
}
```

10
20
30
40
50

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

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

for-loop

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

10
20

for-each

```
for (int[] x : a)
{
  for (int y : x)
  {
    S.o.pln(y);
  }
}
```

10
20
30
40
50

→ Even though for-each loop is more 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

(1) within the switch to stop fall through

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

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

Ex:-

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

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

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

Op:-
Hello
End

→ if we are using break statement any where else we will get Compiletime Error

Ex:-

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

C.E break outside switch or loop.

Continue Statement:-

→ we can use continue statement to skip current iteration and continue for the next iteration inside loops

Ex:-

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

1
3
5
7
9

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

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

labeled break & Continue Statements:-

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

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

Ex 2:-
```
l1:
for(int i=0; i<3; i++)
{
  for(int j=0; j<=3; j++)
  {
    if(i==j)
      break;
    S.o.pln(i+"-----"+j);
  }
}
```

break:-
```
1------0
2------0
2------1
```

break l1:-

No output

Continue:-
```
0----1
0----2
1----0
1----2
2----0
2----1
```

Continue l1:-
```
1....0
2----0
2----1
```

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

Ex:-

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

1
4
6
8
10

x=1
(i)
0
1<5
x=01
1
x 5<5
3<10
4
x++
6
8
7<10
8
9
10
1

Imp Note:-

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

Ex:- ①
```
if(true)
{
S.o.pln("Hello");
}
else
{
S.o.pln("Hi");
}
```

O/P:- Hello

②
```
while(true)
{
S.o.pln("Hello");
}
S.o.pln("Hi");
```

O/P:- C.E:- unreachable statement

etmSaiRamprabu

Declarations & Access Modifiers

① Java Source file Structure (1 - 9) Packages - 8

② Class modifiers (10 - 14)

③ member modifiers (15 - 23)

* ④ Interfaces. (24 - 31)

Java Source file Structure :-

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

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

Ex :-
```
Class A
{
}
Class B
{
}
Cass C
{
}
```

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

Case(1):-

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

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

Case2 :→

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

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

Ex:-

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

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

Save ⇒ Durga.java

javac Durga.java

A.class B.class C.class D.class

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();
 }
}
```

→ Where even we are using import statement it is not required to use fully qualifiedname hence it reduces improves readability & reduces length of the code.

Case(1):-

Types of import statements:-

↳ There are 2 types of import statements

(1) Explicit class import
(2) Implicit class import

import statements

Explicit class import:-

Ex:- import java.util.AL;

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

→ Best suitable for Hitech City where readability is 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 Ameerpet 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 are not. Hence this is "Static loading".

→ But in the case of java language import statement no class file will loaded at the time of import statement, in the next lines of code when ever we are 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?

✗ ① import java.util;

✗ ② import java.util.AL.*;

✓ ③ import java.util.*;

✓ ④ import java.util.AL;

Case 4:

→ Consider the code,

```
class MyRemoteObject extends java.rmi.UniCastRemoteObject
{
}
```

→ The code compiles fine eventhough 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 :-

Date, List } available in both util, sql

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

C.E :- " Reference to Date is 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
{
  p.s.v.m(String[] args)
  {
    Date d = new Date();
  }
}
```

order :-

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

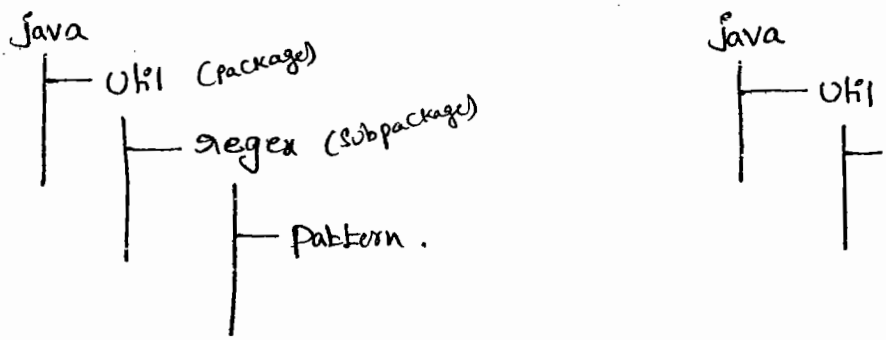
Conclusion :- While Resolving class names Compiler will always gives the precedence in the following order,

→ order see above

Case 7 :-

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

Ex :-



→ To use PatternClass which of the following import is required

✗ ① import java.*;

✗ ② import java.util.*;

✓ ③ import java.util.regex.*;

✓ ④ import java.util.regex.Pattern;

Case (8) :-

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

① java.lang package.

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

Case 9 :-

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

Static import :-

→ This Concept introduced in 1.5 Version.

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

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

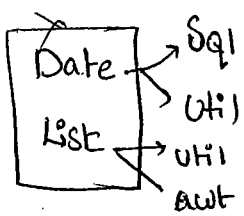
ex:-

Without Static import

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

With Static import

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



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

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

- Test → It is a class name
- name → Static variable present in Test class of the type String
- length() → It is a method present in String class

```
class System
{
static PrintStream out;
}
System.out.println();
```

- System → It is a class present in java.lang package
- out → It is a static variable of type PrintStream present in System class
- println() → It is a method present in PrintStream class.

Explanation :-

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

→ But when even 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 prospective (Ambiguity):-

Ex:-
```
import static java.lang.Integer.*;
import static java.lang.Byte.*;
Class Test
{
  p.s.v.m(String[] args)
  {
    S.o.pln(MAX_VALUE);
  }
}
```

C.E:- reference to MAX_VALUE is ambiguous

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

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

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

Ex:-

```
import static java.lang.Integer.MAX_VALUE;  ---> (2)
import static java.lang.Byte.*;  ---> (3)
Class Test
{
  Static int MAX_VALUE =999;  ---> (1)
  P.S.V.m(String[] args)
  {
    S.O.pln(MAX_VALUE);
  }
}
```

→ If we are Commenting Line(1) then Explicit Static import will get priority Hence we will get Integer class MAX_VALUE is o/p 2147483647.

→ If we are Commenting Lines(1) & (3) then Byte Class MAX_VALUE will be Considered & we will get 127 as o/p.

(-ve point):-

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

Q) Which of the following import Statements are valid.

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

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

✗ ③ import static java.lang.math;

✓ ④ import java.lang.math;

✓ ⑤ import static java.lang.math.*;

✗ ⑥ import static java.lang.math.sqrt();

→ problems

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

Normal 'import' Vs Static import:-

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

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

Packages

9346 classes are their in Java according to 1.6v

Package:-

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

① To resolve naming Conflicts.

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

③ It improves modularity of the application.

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

Com.icicibank.loan.housingloan.Account

- Com.icicibank → domain name in reverse
- loan → module name
- housingloan → Submodule name
- Account → Class name.

Ex:-

```
Package Com.durgajobs.itjobs;

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

① javac HydJobs.java

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

CWD
|→ HydJobs.class.

② javac -d . HydJobs.java

→ destination → to place generated class files

. → current working directory

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

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

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

→ As the destination we Can use any valid directory

Ex: javac -d c: HydJobs.java

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

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

Ex: javac -d z: HydJobs.java

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

Run

→ Java Com.durgajobs.itjobs.HydJobs

O/P :- Getting Job is Very easy.

Conclusions:-

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

Ex :- ✓ package pack1;

→ package pack2;

Class A

{

}

C.E :- class, interface or enum expected.

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

Ex :- ✓ import java.util.*;

→ package pack1;

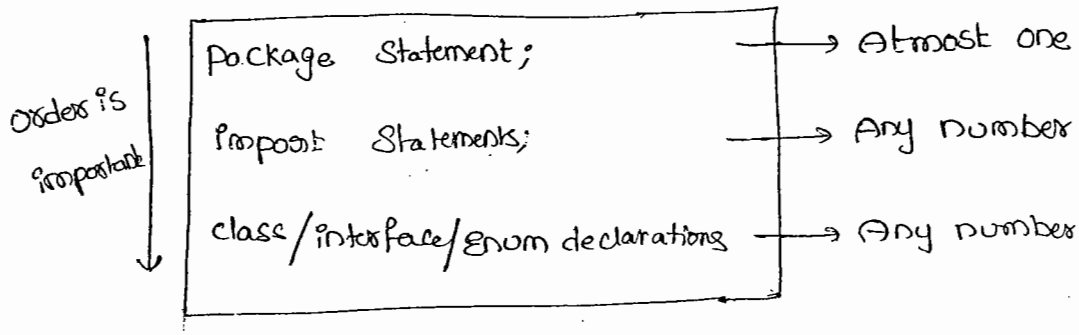
Class A

{

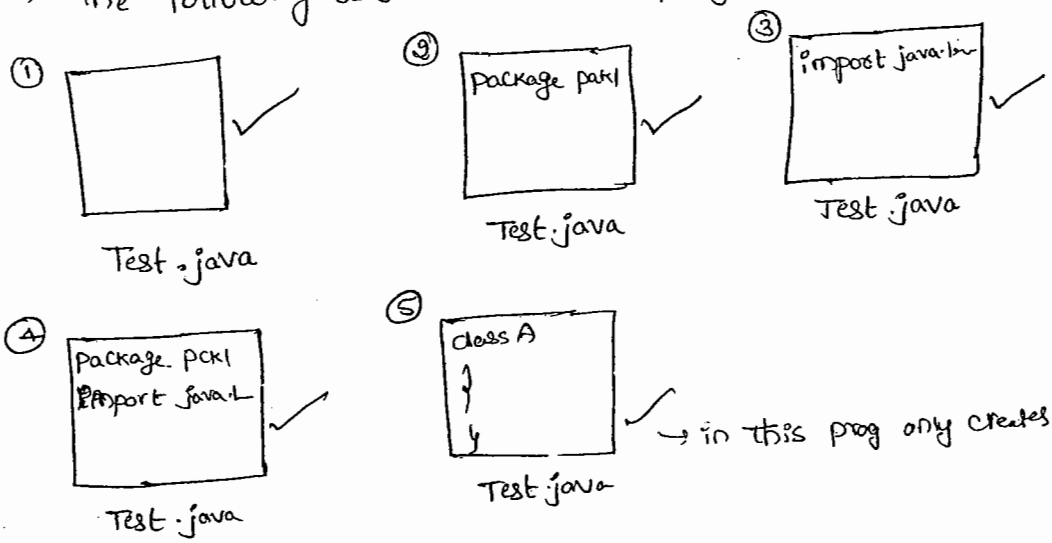
}

C.E :- Class, interface or enum Expected.

→ The proper Structure of a Java Source file is



→ The following are Valid Java programs.



→ An Empty Source file is a Valid Java program.

** Class modifiers **

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

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

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

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

→ we can specify this information by declaring with appropriate modifier.

→ The only applicable modifiers for top-level classes are

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

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

Ex:-

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

C.E:- modifier private not allowed here

→ But for the Inner classes the following modifiers are allowed

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

```
11: private class A
12: {
13: }
14: static class B
15: {
    }
    p.s.v.m(c) →
    {
      S.o.pln("Hi");
    }
```

because the main class is not declare

28/04/11

access Specifiers Vs access modifiers :-

→ In old languages like C & C++ public, private, protected & default are considered as access Specifiers.. & all the remaining like final, Static are considered as access modifiers.

→ But in Java there is no such type of division all are considered as access modifiers.

Public classes :-

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

Ex:-

```
package pack1;
Public class A
{
 Public void m1()
 {
  S.o.pln("Hello");
 }
}
```

Javac -d . A.java

```
CWD
 |— pack1
      |— A.class
```

```
package pack2;
import pack1.A;
class B
{
  p.s.v.m(String[] args)
  {
      A a = new A();
      a.m1();
  }
}
```

Compl. javac -d . B.java ↵

Run: java pack2.B ↵

→ If we are not declaring class A as public, then we will get Compile-time Error while 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 with in that current package. i.e from outside of the package we can't access.

final modifier :-

→ final is the modifier applicable for classes, methods & variables.

→ If a method declared as the final then we are not allowed to ~~to~~ override that method in the child class.

ex:-

```
Class P
{
  public void property()
  {
    S.o.pln("money + Gold + Land");
  }
  public final void marry()
  {
    S.o.pln("Subba laxmi");
  }
}
Class C extends P
{
  public void marry()
  {
    S.o.pln("kajal | 3sba | atara");
  }
}
```

C.E

C.E:- marry() in C cannot override marry() in P; overridden method is final.

→ If a class declared as the final then we can't create child class

ex:-

```
final Class P
{
}
```

```
Class C extends P
{
}
```

X

Ex:-
```
final class P
{
}
Class C extends P
{
}
```

C.E:- Can't inherit from final P.

→ Every method present inside a final class is always final bydefault. but every variable present in final class need not be final.

→ The main Advantage of final keyword is we can acheive Security as no one is allowed to change our implementation.

→ But the main disAdvantage of final keyword is we are missing key benefits of OOP's Inheritance & polymorphism (overriding). Hence, if there is no specific requirement never recommended to use final keyword.

** abstract modifier :-

→ abstract is the modifier applicable for classes & methods but not for variables.

abstract method :-

→ Eventhough we dont ~~about~~ know about implementation still we can declare a method with abstract modifier. i.e abstract methods can have only declaration but not implementation. Hence, Every abstract method declaration should compulsary ends with ";".

Ex:-

✗ 1) public abstract void m1() { }

✓ 2) public abstract void m2();

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

Ex:-

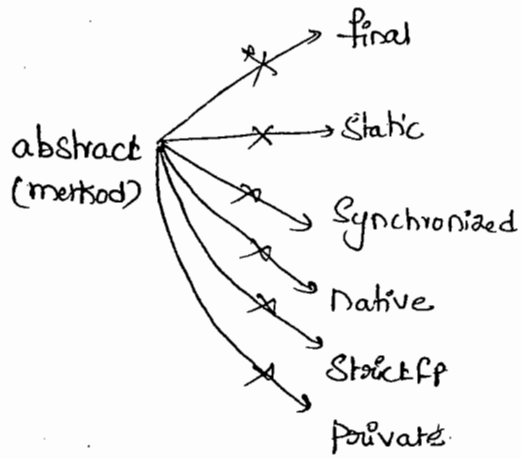
```
abstract class Vechicle
{
  public abstract int getNoOfWheels();
}

Class Bus extends Vehicle
{
  public int getNoOfWheeels()
  {
    return 6;
  }
}

Class Auto extends Vehicle
{
  public int getNoOfWheels()
  {
    return 3;
  }
}
```

→ By declaring abstract methods in parent class we can define Guidelines to the child classes which describes the methods those are to be Compulsary implemented by child class.

29/04/11

→ abstract modifier never talks about implementation, if any modifier talks about implementation then it is always illegal combination with abstract.

→ The following are various illegal combinations of modifiers for methods



abstract class :-

→ for any java class if we don't want instanciation then we have to declare that class as abstract. i.e, for abstract classes instanciation (creation of object) is not possible.

Ex:-

```
abstract class Test
{
}
Test t = new Test();
```

C.E:- Test is abstract; cannot be instantiated

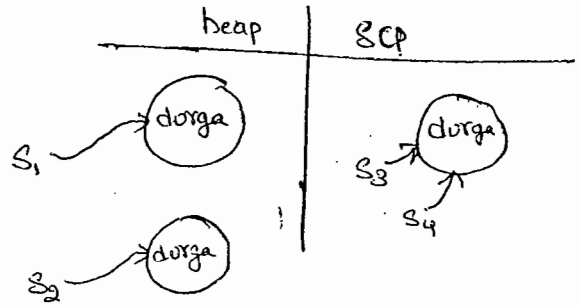
```
Test t = new Test();
```

Ex:- String S_1 = new String("durga");

String S_2 = new String("durga");

String S_3 = "durga";

String S_4 = "durga";



otm SaiRamprabu.

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

durga Sir.

✓ 1) Language fundamentals
✓ 2) Operators & Assignments
3) Flow-Control
4) declaration & Access modifier
5) oops concepts
✓ 6) Exception Handling
7) multi threading
✓ 8) Inner classes
9) java.lang package
✓ 10) java.io.package
11) Serialization
✓ 12) java.util.package (Collection frame works)
13) Generics
14) Regular Expressions
✓ 15) G.C
✓ 16) Assertions (1.4)
17) I18N
18) enum
19) developement.

Dell

chaitanyaobs@gmail.com
Satish4dworld@ "

29444524

Chait

Chaithanya_Anumanchi@dell.com.

ON durgajobsInfo 9870807070

abstract class Vs abstract method :-

→ If a class Contains atleast One abstract method then Compulsary That class should be declared as abstract otherwise we will get Compiletime Error. because, the implementation is not Complete & hence We Can't Create an object.

→ Eventhough Ea class doesnot Contain any abstract method still we Can declare the class as abstract. i.e, abstract class Can Contain Zero "0 no. of abstract method.

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

Ex:-

① Class Test
{
public void m1();
}
C.E:- missing method body, or declare abstract

② class Test
{
public abstract void m1()
{
}
}
C.E:- abstract methods Can't have a body

③ Class Test
{
public abstract void m1();
}

C.E:- Test is not abstract and doesn't override abstract method m1() in Test.

Ex-4:-

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

C.E:- SubTest is not abstract and does not override abstract method m2() in Test

→ We can handle these Compiletime Errors either by declaring class SubTest as abstract or by providing implementation for m2().

Note:-

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

abstract Vs final :-

→ abstract methods we have to override in child classes to provide implementation. where as final methods can't be overridden. Hence, abstract final combination is illegal combination for methods.

→ for abstract classes we should create child classes to provide proper implementation but for final classes we can't create child class. Hence abstract final combination is illegal for classes.

→ final class Can't have abstract methods where as abstract class Can Contain final methods.

```
final class A
{
    abstract void m1();
}
```
X

```
abstract class A
{
    public final void m1()
    {
    }
}
```
✓

Strictfp (all lower case) modifier :- (strictfloatingpoint)

→ Strictfp is the modifier applicable for methods & classes but not for variables.

→ if a method declared as strictfp all floatingpoint Caluclations in that method has to follow IEEE 754 Standard So, that we will get platform independent results.

→ Strictfp method always talks about implementation where as abstract method never talks about implementation. Hence strictfp-abstract method Combination is illegal Combination for methods.

→ If a class declared as strictfp then every Concreate method in that class has to follow IEEE 754 Standard so, that we will get platform independent results.

→ abstract-strictfp Combination is legal for classes but illegal for methods

ex:- abstract strictfp class Test
{
}
✓

public abstract strictfp void m1(); ✗ (invalid).

Member (variables & methods) modifiers :-

① public members :-

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

ex:-

```
Package pack1;

→ Class A
  {
    public void m1()
    {
      S.o.pln("Hi");
    }
  }
```

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

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

② default members :-

→ If a member declared as the default, then we can access that member only with in the current package & we can't access from outside of the package. Hence, default access is also known as package level access.

③ private members :-

→ If a member declared as private then we can access that member only within the current class.

→ abstract methods should be visible in child classes to provide implementation. where as private methods are not visible in child classes. Hence private-abstract combination is illegal for methods.

④ protected members : (The most misunderstood modifier in Java) :-

→ If a member declared as protected then we can access that member with in the current package anywhere but outside package only in child classes.

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

*→ with in the current package we can access protected members either by parent reference or by child reference.

→ But from outside package we can access protected members only by using child reference. if we are trying to use parent reference we will get C.E

Ex:.

```
package pack1;
public class A
{
  protected void m1()
  {
    S.o.pln("the most misunderstood modifier in Java");
  }
}
Class B extends A
{
  P.S.v.m(——)
  {
```

✓ A a = new A()

✓ a.m1();

✓ B b = new B()

✓ b.m1();

✓ A a1 = new B()

✓ a1.m1();

```
package pack2;
import pack1.A;
public class C extends A
{
  p.s.v.m(——)
  {
    A a = new A()
  ✗ a.m1();
    C c = new C()
  ✓ c.m1();
    A a1 = new C()
  ✗ a1.m1();
  }
}
```

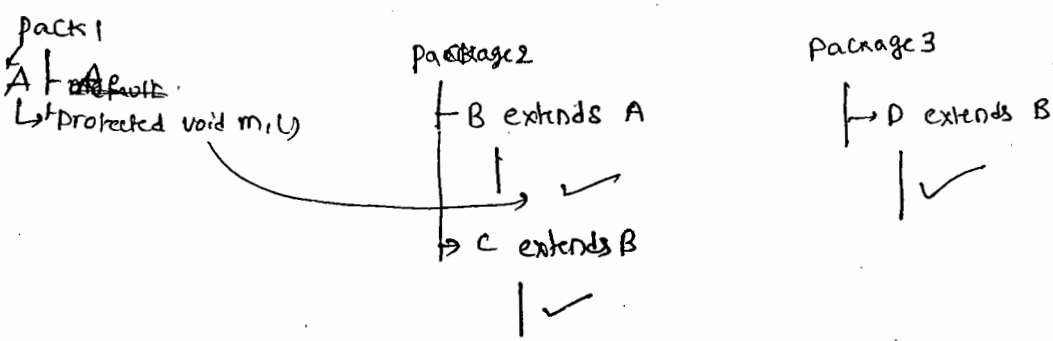
→ The most restricted modifier is "private"

→ The most accessible modifier is "public"

→ private < default < protected < public

→ The recommended modifier for variables is private

→ The recommended modifier for methods is public



→ The most restricted

* private < default < protected < public

Visibility	private	<default>	protected	public
① with in the Same Class	✓	✓	✓	✓
② from child class of Same Package	✗	✓	✓	✓
③ from non-child class of Same package	✗	✓	✓	✓
④ from child class of outside Package.	✗	✗	✓ (But we should use only child class reference)	✓
⑤ from non-child class of outside package.	✗	✗	✗	✓

"final" variables :-

→ In General for instance & static variables it is not required to perform initialization Explicitly JVM will always provide default Values.

→ But for the local variables JVM won't to provide any default Values Compulsary 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 Compulsary we should perform initialization wheather we are using or not otherwise

we will get Compiletime Error

ex:-

```
Class Test
{
  int x;
}
```

```
Class Test
{
  final int x;
}
```

C.E:- variable x might ~~have~~ not have been initialized.

Rule:-

(***) for the final instance variables we should perform initialization before Constructor Complition.

→ i.e, the following are various places for this,

① At the time of declaration

ex:-

```
class Test
{
  final int x=10;
}
```

② inside instance Block.

ex:-

```
Class Test
{
  final int x;
  {
    x=10;
  }// instance Block
}
```

③ inside Constructor.

ex:-

```
Class Test
{
  final int x;
  Test()
  {
    x=10;
  }
}
```

→ Other than these if we are perform initialization any where else we will get Compiletime Error.

Ex:-

```
Class Test
{
    final int x;
    public void m1()
    {
        x=10;   ✗
    }
}
```

C.E:- Cannot assign a value to final variable x.

final Static Variables:-

→ for the normal static variables it is not required to perform initialization explicitly, JVM will always provide default values.

→ But for final static variables we should perform initialization explicitly otherwise we will get C.E.

Ex:-

```
Class Test
{
    Static int x;
}
```

✓

```
Class Test
{
    final static int x;
}
```

✗

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

Rule:-

* for the final static variables we should perform initialization before class loading complition.
* i.e, the following are various places to perform this

(1) At the time of declaration

Eg:- Class Test

```
{
    final static int x=10;
}
```

(2) Inside Static Block

Eg:- Class Test

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

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

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

C.E:- Can't assign a ~~variable~~ value to final variable x.

iii) final Local variables :-

→ for the local variables JVM won't to provide any default values Compulsary we should perform initialization before using that variable.

Ex:- ①

```
Class Test
{
    public static void main()
    {
        int x;
        S.o.pln("Hello");
    }
}
```

O/P :- Hello

②

```
Class Test
{
    public static void main()
    {
        int x;
        S.o.pln(x);
    }
}
```

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

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

Ex:

```
Class Test
{
    P.S.V.M()
    {
        final int x;
        S.o.pln("Hello Sai");
    }
}
```

O/P :- Hello Sai.

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

ex:-

```
Class Test {
    P.S.V.M() {
        public int x=10;
        private int x=20;
        static int x=60;
        protected int x=30;
        final int x=40;
    }
}
```

→ formal parameters of a method simply access as local variables of that method. hence, a formal parameter can be declared as final.

→ If we declare a formal parameter as final within the method we can't change its value otherwise we will get Compiletime Error.

Ex:-

```
Class Test
{
  p.s v.m(——)
  {
     m1(10, 20);
  }
  p.s.v.m1(final int x, int y)
  {
     x=1000;   // Can't assign a value to final variable x.
     y=2000;

     S.o.pln(x+"----"+y);
  }
}
```

Actual parameters

formal parameters

Static → class level

instance → object level

Static modifier :-

→ Static is the modifier applicable for variables & methods but not for classes (but innerclass can be declared as static).

→ if the value of a variable is varied from object to object then we should go for instance variable. In the case of instance variable for every object a seperate copy will be created.

→ If the value of a variable is same for all objects then we should go for static variables. In the case of static variable only one copy will be created at class level and share that copy for every object of that class.

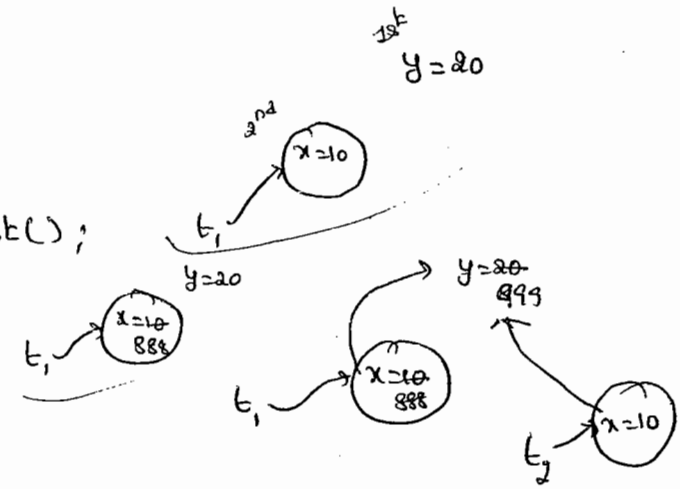
first Static variable is created at the begining when class is created.

19

Ex:-

```
Class Test
{
  int x=10;
  Static int y=20;
  P.S.V.m(——)
  {
    Test t1 = new Test();
    t1.x = 888;
    t1.y = 999;
    Test t2 = new Test();
    S.o.pln(t2.x + "---" + t2.y);
    }
}
```

10 999



for every object a Seperate copy will be Created.

→ Static members can be accessed from both instance & Static areas where as instance members can be accessed only from instance area directly. i.e, from static area we can't access instance members directly otherwise we will get Compiletime Error.

Q) Consider the following declarations

I. int x=10;

II. Static int x=10;

III. Public void m1()
```
{
  S.o.pln(x);
}
```

IV. Public Static void m1()
```
{
  S.o.pln(x);
}
```

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

✓ A). I & III

✗ B). I & IV . CE :- non-static variable x cannot be accessed from static context

✓ C) II & III

✓ D) II & IV

✗ E) I & II

✗ F) III & IV

→ for static methods compulsary implementation should be available where as for abstract methods implementation should not be available Hence abstract-static combination is illegal for methods.

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

Ex :-

```
Class Test
{
 p. s. v. m(String[] args)
 {
   S.o.pln(" String[] ");
 }
 public static void main(int[] args)
 {
   S.o.pln(" int[] ");
 }
}
```

O/P :- String[]

→ ~~But JVM also~~

→ But Jvm always Call String arrguements main method only. The other main method we have to Call explicitly Just Like a normal method Call.

→ Inheritance Concept is applicable for static methods including main() method hence while executing child class if the child doesnot Contain main method then the parent Class main method will be execute

ex:-

```
Class p
{
  p.s.v.m(String[] args)
  {
     S.o.pln("parent Class");
  }
}
Class c extends p
{
}
```

Javac p.java

P.class C.class

o/p Java p
Parent class

o/p Java C
parent class.

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

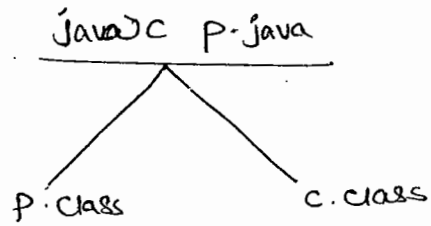
ex:-

```
Class p
{
  p.s.v.m(——)
```

Ex:-

```
class P
{
  p.s.v.main(String[] args)
  {
    S.o.pln("parent classmain");
  }
}
class C extends P
{
  p.s.v.main(String[] args)
  {
    S.o.pln("child metin");
  }
}
```

it is not overriding
it is method hiding



java P ↵

parent main

java C ↵

child main.

native modifier :-

→ native is the modifier applicable only for methods but not for variables and classes.

→ The native methods are implemented in some other languages like C & C++ hence native methods also known as "foreign methods".

→ The main objectives of native keyword are ① to improve performance of the system

① To improve performance of the system.

② To use already existing legacy non-Java code.

Pseudo Code :-

→ To use native keyword

ex:-

```
class Native
{
  static
  {
    System.loadLibrary("native library")
  }
  public native void m1();
}

class Child
{
  p.s.v.m(——)
}

Native n = new Native();
n.m1();
}}
```

① Load native library

② Declare a native method

③ Invoke a native method

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

Ex:- ① class Test

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

X

C.E :- native methods can't have a body.

② public native void m1();

① → for native methods implementation should be available in some other languages where as for abstract methods implementation should not be available hence abstract-native combination is illegal combination for methods.

② → native methods cannot be declared with strictfp modifier because there no guarunte 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.

"Synchronized" modifier :-

→ Synchronized is the modifier applicable for methods & Blocks. we Can't declare class & variable with this Keyword.

→ If a method (or) Block declared as Synchronized then at a time only one Thread is allowed to operate on the given Object.

→ The main advantage of Synchronized Keyword is We Can resolve data inconsistancy problems. But the main dis-Advantage of Synchronized keyword is it increases waiting time of thread and effects performance of the System. Hence, If there is no Specific requirement it is never recommended to use Synchronized Keyword.

"transient" modifier :-

→ transient is the modifier applicable only for Variables & we Can't apply for methods & classes.

→ At the time of Serialization, if we don't want to Save the value of a Particular variable to meet Security Constraients, then we should go for transient Keyword.

→ At the time of Serialization JVM ignores the original value of transient variable & default value will be Serialization.

"Volatile" modifier :-

→ Volatile is the modifier applicable only for variables but not for methods & Classes.

→ If the value of a variable keep on changing Such type of variables we have to declare with volatile modifier.

- If a variable declared as volatile then for every thread a seperate local copy will be created.
- Every intermediate modification performed by that thread will takes place in local copy instead of master copy.
- Once the value gets finalized just before terminating the thread the master copy value will be updated with local stable value.
- The main advantage of volatile keyword is we can ~~achieve~~ resolve data inconsistency problems.
- But the main disadvantage of volatile keyword is, creating & maintaining a seperate copy for every thread, increases complexity of the programing & effects performance of the system. Hence, if there is no specific requirement it is never recommended to use volatile keyword, & it is almost outdated keyword
- Volatile variable means it's value keep on changes where as 'final' variable means its value never changes. Hence final-volatile combination is illegal combination for variables.

Conclusion:

→ The only applicable modifier for local variables is final

→ The modifiers which are applicable only for variables, but not for classes & methods are. Volatile & transient

→ The modifiers which are applicable only for methods but not for classes & variables native & synchronized.

→ The modifiers which are applicable for top level classes, methods & variables are public, <default>, final

Modifier	Classes		methods	variables	blocks	interfaces	enum	Constructors
	Outer	Inner						
public	✓	✓	✓	✓	×	✓	✓	✓
<default>	✓	✓	✓	✓	×	✓	✓	✓
private	×	✓	✓	✓	×	×	×	✓
protected	×	✓	✓	✓	×	×	×	✓
final	✓	✓	✓	✓	×	×	×	×
abstract	✓	✓	✓	×	×	✓	×	×
static	×	✓	✓	✓	✓	×	×	×
synchronized	×	×	✓	×	✓	×	×	×
native	×	×	✓	×	×	×	×	×
strictfp	✓	✓	✓	×	×	✓	✓	×
transient	×	×	×	✓	×	×	×	×
volatile	×	×	×	✓	×	×	×	×

→ The modifiers which are applicable for Inner classes but not for Outer Classes are private, protected, Static

Interfaces

(1) Introduction

(2) Interface declaration & Implementation
  (a) extends Vs implements.

(3) Interface methods

(4) Interface Variables

(5) Interface Naming Conflicts
  (1) method Naming Conflicts
  (2) Variable " "

(6) marker Interface

(7) Adapter class

(8) Abstract Class Vs Concrete Class vs interface.

(9) diff. b/w abstract class & interface

Interface :-

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

→ from the client point of view ~~can~~ 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.

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

Ex:-

→ By using Bank ATM GUI Screen, Bank people will highlate the Set of Services what they are offering At the Same time the Same Screen describes the Set of Services what End-User is Expected. Hence this GUI Screen acts as Contract b/w ~~the~~ bank people & customers.

→ with in the Interface we Can't write any implementation because it has to highlight Just the Set of Services what we are offering or what you are Expecting. Hence every method present inside interface should be abstract. Due to this interface is Considered as 100% pure abstract class

What is an Interface :-

→ Any Service requirement Specification (SRS) (or) Any Contract b/w Client & Service provider (or) 100% pure abstract class is nothing but an Interface.

→ The main Advantages of Interfaces are:

(i) we can acheive Security. because we are not highlighting our internal implementation.

(ii) Enhancement will become very easy, because 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 compulsary we should provide implementation for every method of that interface otherwise we have to declare class as abstract. Violation leads to Compile-time Error.

→ when ever we are implementing an interface method compulsary it should be declared as public otherwise we will get CompileTimeError.

Extends Vs implements :-

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

ex:-

```
interface A
{
}
interface B
{
}
interface C extends A, B
{
}
```

Q) which of the following is True?

(1) A class can extend any no. of classes at a time. ✗

(2) A class can implement only one Interface at a time. ✗.

(3) A class can extend a class ~~and~~ or can implement an interface but not both simultaneously ✗

(4) An Interface can extend only one interface at a time ✗

(5) An Interface can implement any no. of classes at a time ✗

(6) none of the above ✓

Q) Consider the expression

$\boxed{X \text{ extends } Y}$ — for which of the following possibilities this expression is True?

① Both should be classes

② Both should be interfaces

✓ ③ Both can be either classes or interfaces

④ No Restriction.

A:

① X extends Y, Z

(a) X, Y, Z should be interfaces

② X extends Y implements Z

X, Y → classes

Z → interfaces

③ X implements Y extends Z → C.E

Interface methods:-

whether we are declaring or not, every interface method is by-default, public & abstract

ex:- interface Interf
{
void m1();
}

public:-

→ To make this method availability for every implementation class.

abstract:-

Because interface methods specifies requirements but not implementation.

Hence the following method declarations are equal inside interface.

(1) void m1(); ✓
(2) public void m1(); ✓
(3) abstract void m1(); ✓
(4) public abstract void m1(); ✓

→ As every interface method is bydefault public & abstract the following modifier are not applicable for interface methods.

(1) private
(2) protected
(3) <default>
(4) final

(5) static
(6) strictfp
(7) synchronized
(8) native

→ which of the following method declaration are valied inside interface:

(1) public void m1() {} ✗

(2) public static void m1(); ✗

(3) public synchronized void m1(); ✗

(4) private abstract void m1(); ✗

(5) public abstract void m1(); ✓

<u>interface variables</u>:-

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

<u>Constants at requirement level</u>:-

→ Every interface variable is always public, static, final whether we are declaring (or) not.

```
interface Inter
{
 int x=10;
}
```

<u>public</u>:- To make this variable available for every implementation class.

<u>static</u>:- without existing object also implementation class can access this variable.

<u>final</u>:- implementation class can access this variable but can't modify.

→ Hence inside interface the following declaration are valid & equal.

1) int x=10;
2) public int x=10;
3) public static int x=10;
4) public static final int x=10;
5) public static int x=10;
6) final int x=10;
7) public final int x=10;

8) Static final int x=10;

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

(1) private (3) <default> (5) volatile.

(2) protected (4) transient

→ for the interface variable compulsary are should perform initialization at the time of declaration only otherwise will get compile time Error.

(*) interface Interf
{
int x; ✗ C.E:- = Expected.
}

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

(1) int x=10; ✓
(2) int x; ✗
(3) private int x=10; ✗
(4) public int x=10; ✓
(5) transient int x=10; ✗
(6) volatile 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;
    S.o.pln(x);
 }
}
```

C.E:

```
Class Test implements Interf
{
 p.s.v.m(String[] args)
 {
    int x=88;
    S.o.pln(x);88
 }
}
```

Interface Nameing Conflicts:-

① method naming Conflicts:-

Case1:-

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

Ex:-

```
interface Left
{
  Public void m1();
}
```

```
interface Right
{
  Public void m1();
}
```

```
Class Test implements Left, Right
{
    public void m1()
    {
    }
}
```

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 returntypes. Then it is impossible to implement both interfaces at a time.

Ex :-

```
interface Left
{
  public void m1();
}
```

```
interface Right
{
  public int m1();
}
```

→ We Cant write any Java Class which implements both interfaces Simotaneously.

Q) is It possible A Java class Can implement any no. of interfaces Simultaneously.

A) yes, Except if Two interfaces Contains a method with Same Signature but different return-types.

(2) Variable naming Conflicts :-

```
interface Left
{
  int x=888;
}
interface Right
{
  int x=999;
}
```

```
Class Test implements Left, Right
{
  p.s.v.m(——)
  {
   S.o.pln(x);
  }
}
```

C.E:- reference to x is abiguous.

→ There may be a chance of 2 interfaces contains variable with same name & may rise variable naming conflicts but we can resolve these naming conflicts by using interface names.

```
S.o.p(Left.x) ; 888
S.o.p(Right.x) ; 999
```

Marker Interface :-

Ex :- Kenya

→ If an interface wont contain any method & by implementing that interface if other objects will get ability such type of interfaces are called marker interface (or) Tag interface (or) ability interface.

Ex:- Serializable, Clonable, Random Access, Single Thread model.

→ these interfaces are marked from some ability.

Ex:- By implementing Serializable interface we can send objects across the N/w and we can save state of object to a file. This extra ability is provided through Serialable interface.

A:- By implementing Cloneable interface our Object will be in a position to provide exactly duplicate Objects

Q) Marker interface cant Contain any method then how the Objects will get that Special ability?

A) JVM is responsible to provide required ability in marker interfaces.

Q) Why JVM is providing required ability in marker interface?

A) To reduce Complexity of the programing.

Q) Is it possible to Create our own Marker Interface?

A) Yes, But Customization of JVM is required.

Ex:- Sleepable, Eatable, Jumpable, Lovable, Funnable.

Adapter Class:-

→ Adapter class is a Simple java class that implements an interface, an interface only with Empty implementation.

```
interface X
{
  m1();
  m2();
  :
  m1000();
}
```

```
abstract class Adapter X implements X
{
  m1() {}
  m2() {}
  :
  m1000() {}
}
```

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

→ If we implement an interface directly ~~coo~~ Compulsary we should provide implementation for every method of that interface, whether we are 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()
  {
    --
  }
}
```

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 Completly (Just partially implementation) then we should go for abstract Class.

Ex:- Generic – Servlet
HTTP_Servlet

→ we are talking about implementation Completly & ready to provide Service, Then we should go for Concrete Class.

Ex:- Our own Servlet.

Difference b/w interfaces & abstract class:-

interface	abstract class
1) If we don't know any thing about implementation Just we have requirement Specification. then we should go for interface.	1) If we are talking about implementation but not Completly (partially implementation) then we should go for abstract class.
2) Every method present inside interface is by default public & abstract.	2) every method present inside abstract class need not be public & abstract. we can take Concrete methods also.
3) The following modifiers are not allowed for interface methods: strictfp, protected, static, native private, final, synchronized,	3) There are no restrictions for abstract class method modifier i.e, we can use any modifier.

4) every variable present inside interface is public, static final, by default whether we are declare (or) not	4) abstract class variables need not be public, final static.
5) for the interface variables we can't declare the following modifiers private, protected, transient, volatile	5) There are no restriction for abstract class variable modifiers.
6) for the interface variables compulsary we should perform initialization at the time of declaration only	6) for the abstract class variables there is no restriction like performing initialization at the time of declaration
7) Inside interface we can't take instance & static blocks.	7) Inside abstract class we can take static block & instance blocks.
8) Inside Interface we can't take constructor.	8) Inside abstract class we can take constructor.

Q) Inside abstract class we can take constructor but we can't create an object of abstract class, what is the need?

A) → abstract class constructor will be executed whenever we are create child class object to perform initialization of parent class instance variable at parent level only and this constructor meant for child object creation only

Q) Inside Interface every method should be abstract where as in abstract class also we can take only abstract methods then what is the need of interface?

A) → Interface purpose we can replace abstract class but it is not a good programming practice we are miss using the role of abstract class.

→ we should bring abstract class into the picture whenever we are talking about implementation.

28/4/11

Om Sai Ram prabu.

Oops Concept

1) Data Hiding 2

2) Abstraction 2

3) Encapsulation 2

4) Tightly Encapsulated class 3

5) IS-A Relationship 3

6) Has-A Relationship 5

7) Method Signature 6

* 8) Over loading 7

9) Over riding 10

10) Method hiding 14

11) Static Control flow 18

12) Instance Control flow. 22

13) Constructors 24

14) Coupling 42

15) Cohesion 43

16) Type-Casting -40

polymorphism — 17

Type-casting - 40

① Data Hiding :-

→ Hiding of the data, So that outside person Cant access our data directly.

→ By using private modifier we Can implement Data Hiding.

Ex:-

```
Class Account
{
  private double balance = 1000;
}
```

→ The main Advantage of Data Hiding is we Can acheive Security.

② Abstraction :-

→ Hiding internal implementation details & just highylate the set of Services what we are offering, is called "Abstraction".

Ex:-

↳ By Bank ATM machine, Bank people will highLate the Set of Services what they are offering without Highlating internal implementation. This Concept is nothing but Abstraction.

→ By Using interfaces & abstract classes we Can acheive abstraction.

→ The main Advantages of Abstraction are.

1) We Can acheive Security as no one is allowed to Know our internal implementation.

2) With out effecting outside person we Can change our internal implementation Hence Enhancement will become Very easy.

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

4) Tightly Encapsulated class :-

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

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

Ex:-

```
Class A
{
  Private int balance;
  Public int getBalance()
  {
    return balance();
  }
}
```

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

✓
```
Class A
{
  Private int x=10;
}
```

✗
```
Class B extends A
{
  int y=20;
}
```

✓
```
Class C extends A
{
  Private int z=30;
}
```

86

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 acheive security.

② Enchancement will become very easy.

③ improves modularity of the application.

Ex 3:- Which of the following classes are Tightly Encapsulated.

Ex:

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

Q) by default what modifier to variables?

Conclusion:-

→ If parent class is not tightly Encapsulated then no child class is Tightly Encapsulated.

5) IS-A Relationship :-

→ It is also known as Inheritance

→ By using extends Keyword we can implement IS-A Relationship

→ The main advantage of IS-A Relationship is Reusability of the Code.

Ex:-

```
Class P
{
  Public void m1()
  {
    ====
  }
}
Class C extends P
{
  public void m2()
  {
}
```

```
Class Test
{
 P.S.V.m(String[] args)
 {
```

Case1: P p = new P();

p.m1(); ✓

p.m2(); X → C.E:- Cannot find Symbol

Symbol : method m2()

location : class P

Case 2: C c = new C();

c.m1(); ✓

c.m2(); ✓

* Case3:. P p1 = new C();

p1.m1(); ✓

p1.m2(); X C.E:

* Case 4:. C c1 = new P(); X C.E:- incompatable types

found : P

required : C

Conclusion(1):-

① what ever the parent class has by default available to the child. Hence ~~and~~ with the on child class reference we can call both parent & child class methods.

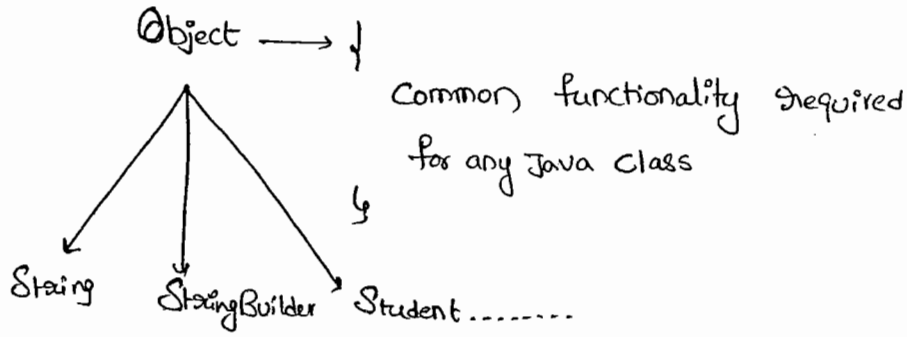
② what ever the child has by default not available to the parent hence on the parent class reference we can call only parent class methods & we cant call child specific methods.

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 it's functionality by default available to every Java classes.



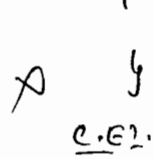
Ex:- the Common functionality which is required for all Exceptions & Errors is defined in Throwable class as Throwable is parent for all Exceptions & Errors, it's functionality will be available automatically to every child not required to rewrite.

Q) Do 'Throwable' has 'Object' as parent class?
→ yes

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

ex:-

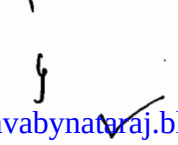
Class A extends B, C



C.E:-

But

interface A extends B C



→ Every class in Java is the child class of Object.

→ If our class doesn't extend any other class then only it is the direct child class of Object.

Ex:- Class A
{
}

Object
↑
A

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

ex:- class A extends B
{
}

Object
↑
B
↑
A
(multi level inheritance).

→ Cyclic inheritance is not allowed in Java

ex:- ① Class A extends B
{
}
Class B extends A ✗
{
}

B
A

C.E:- cyclic inheritance involving A

② Class A extends A
{
}

6) Has - A Relationship :-

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

→ There is no specific keyword to implement Has-A Relationship the mostly we are using "new keyword".

→ The main advantage of Has-A Relationship is Reusability or (code Reusability)

ex:-

```
Class Car
{
  Engine e = new Engine();
}
```

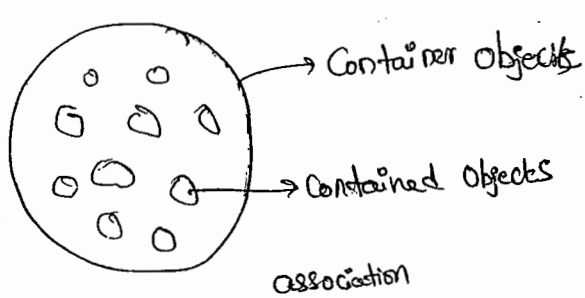
```
Class Engine
{
  // Engine Specific functionality
}
```

Class Car has Engine reference.

→ The main disadvantage of Has-A Relationship is it increases dependency b/w the classes and creates maintaince problems.

Composition Vs Aggregation :-

→ In the case of Composition whenever Container objects is destroyed all Contained objects will be destroyed automatically. i.e, without Existing Container object there is no chance of existing contained object i.e Container & Contained objects having strong association



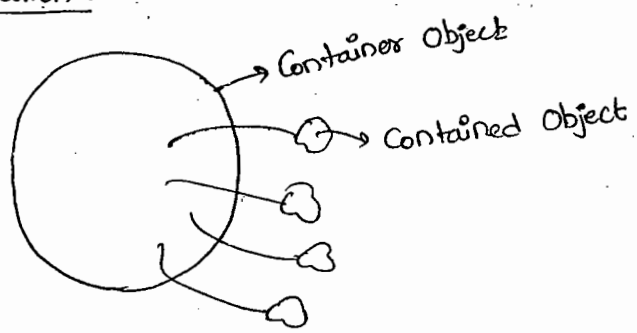
ex:-

⇒ University is Composed of Several departments.

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

Aggregation :-

→ whenever Container object destroyed, there is no garanenty of destruction of Contained objects i.e, without existing Container object there may be a chance of existing Contained object. i.e, Container object just maintains References of Contained objects. This relationship is Called weak association which is nothing but "Aggregation".



ex:-

→ Several proofesors will work in the department

→ whenever we are closing the department still there may be a chance of existing proofesers. The relationship b/w department & professor is Called weak association which is nothing but Aggregation.

```
public void m1(int i)
{
  S.o.pln ("int-arg");
}
public void m1(float f)
{
  S.o.pln ("float-arg");
}
p.s.v.m (———)
{
  Test t = new Test();
      t.m1();    // no-arg
      t.m1(10);  // int-arg
      t.m1(10.5f); // float-arg
}
}
```

*→ In overloading method resolution always takes care by compiler based on reference type. Hence overloading is also considered as compiletime polymorphism (or) Static polymorphism (or) Early binding

→ In overloading reference type will play very important role & runtime object will be dummy.

Case1:-

*Automatic prammotion in overloading:-

→ In overloading method resolution, if the matched method with specified arguement type is not available then compiler doesn't raise

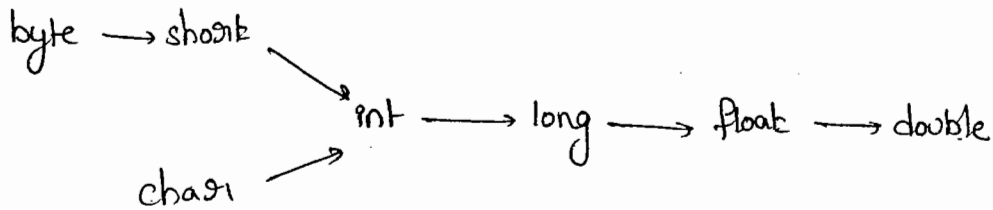
any Error immediately. first it promotes that arrguement to the next Level and checkes for matched method.

→ If the matched method is available then it will be Considered and if it is not available then Compiler once again promotes this arrguement to the next Level.

→ This process will be Continued untill all possible promotions after Completing all promotions Still if the matched method is not available then only we will get C.E.

→ this ~~fol~~ is Called Automatic promotion in overloading.

→ The following are Various possible promotions in Overloading.



Case 1:-

Ex:-

```
Class Test
{
  public void m1(int i)
  {
    S.o.pln("int-arg");
  }
  public void m1(float f)
  {
    S.o.pln("float-arg");
  }
  P.S.V.m(String[] args)
  {
    Test t = new Test();
```

Method Signature :-

→ method Signature Consists of name of the method & argument-List.

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

⇓

m1(int, float)

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

→ Compile will always use method Signature while resolving method Calls

→ with in the Same Class Two methods with the Same Signature not allowed. Otherwise we will get Compiletime Error.

Ex:-
```
class Test
{
  public void m1(int i)
  {
  }
  public int m1(int i)
  {
    return 10;
  }
}
```

m1(int) is the method signature.

```
Test t = new Test()
t1.m1(10);
```

C.E :- m1(int) has already defined in Test

Overloading

Overloading:-

→ Two methods are Said to overloaded iff method names are Same but arguements are different.

→ Lack of overloading in 'C' increases Complexicity of the program. In C, language if there is a change in method arguement type Complusary we should go for new method name.

Ex:- abs() ⟶ int
labs() ⟶ long
fabs() ⟶ float
≡

→ But in Java Two methods having the Same name with different arguements is allowed & these methods are Considered as overloaded methods.

Ex:- abs(int)
abs(long)
abs(float)
≡

→ Having overloading Concept in Java Simplifies the programming

Ex:-
```
Class Test
{
  Public void m1()
  {
    S.o.pln("no-arg");
  }
```

Case:-
→ In overloading more specific version will get highest priority.
what does it mean?

Case 2:-

ex:-

```
Class Test
{
  Public void m1(StringBuffer Sb)
  {
    S.o.pln("StringBuffer-arg");
  }
  Public void m1(String s)
  {
    S.o.pln("String-version");
  }
  public s.v.m(——)
  {
    Test t = new Test();
    t.m1(new SB("durga"));  // StringBuffer-args
    t.m1("durga");          // String version
    t.m1(null);             // C.E:- reference m1() is ambiguity.
  }
}
```

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

```
t.m1('a');  // int-arg
t.m1(10l);  // float-arg
t.m1(10.5); x c.e:-
}                    Cannot find Symbol
}                    Symbol : method m1(double)
                     location : class Test
```

Case 2:-

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

Ex:-

```
Class Test
{
  ①  public void m1(Object o)
     {
       S.o.pln("Object version");
     }
  ②  public void m1(String s)
     {
       S.o.pln("String version");
     }
     p.s.v.m(—)
     {
       Test t = new Test();
       t.m1(new Object());  // object-version
       t.m1("durga");       // String-version
       t.m1(null);          // String-version
     }
}
```

Object
↑
String

suppose ② statement takes // the o/p is object

→ Hence overridding is also known as "runtime polymorphism (or) dynamic polymorphism (or) Late binding".

→ Overriding method resolution is also known as "Dynamic method dispatch".

Rules for Overridding :-

① In overriding method names & arguements must be matched i.e, method signatures must be matched.

② In overriding return type must be matched, But this rule is applicable untill 1.4 version, from 1.5 version onwards Co-variant (ఒక్కటి బదులు మరొకటి) return types are allowed. according to this, child method return-type need not be same as parent method return-type. its child classes also allowed.

Ex:

```
Class P
{
  Public Object m1()
  {
    return null;
  }
}
Class C extends P
{
  Public String m1()
  {
    return null;
  }
}
```

It is valid in 1.5v, But invalid in 1.4v

Ex :-

parent method return type	Object	Number	String	double
	↓	↓	↓	↓
child method return types	String ✓	Integer ✓	Object ✗	int ✗

→ Co-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 overridding concept is not applicable for private methods.

⑤ → Based on our requirement we can declare the same parent class private method in child class also it is valid but it is not overriding.

ex:-

```
Class P
{
  private void m1()
  {
    =
  }
}
Class C extends P
{
  private void m1()
  {
    =
  }
}
```

not overriding

(*) Compiles fines but not overriding.

→ for parent class abstract methods we should overide in child class to provide implementation.

(*) → we can override parent class non-abstract method as abstract in child class to stop parent class method implementation availability to the child classes.

ex):-

```
class P
{
  public void p()
  {
  }
}
abstract class C extends P
{
  public abstract void p();
  }
}
```

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

① native

② Synchronized

③ strictfp

	P: final	non-final	abstract	Synchronized	native
C:	↓ ✗ non-final	↓ ✓ final	↓ ✓ ↑ non-abstract	↓ ✓ ↑ non-Synchronized	↓ ✓ ↑ non-native

→ Strictfp

↓ ✓ ↑

non-strictfp

→ while overriding we can't decrease scope of the modifier but we can increase the following are various acceptable overridings

Private < default < protected < public

p)	Public	protected	default	private
	↓	↓	↓	↓
c)	public.	protected/public	default/protected/public	Private method can't be override

Eg:-

```
Class P
{
 public void m1() { }
}
Class C extends P
{
 protected void m1() X
}
}
```

C.E:-

m_1 in C can't override in C

→ This rule is applicable while implementing interface methods also.

→ when ever we are implementing any interface method compulsary it should be declared as public. because every interface method is public by default.

Ex:-

```
interface Interf
{
  void m1();
}
Class Test implements Interf
{
  void m1()   X C.E:-
}
}
```

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

→ If child class method throws some checked Exception then compulsary Parent class method should throw the same checked Exception or its Parent class Exception. otherwise we will get C.E.

→ But there is no rule for unchecked Exception.

Ex:- ①

```
Class P
{
  Public void m1()
  {
  }
}
Class C extends P
{
  Public void m1() throws Exception
  {
  }
}
```

C.E:- m1() in C can't override m1() in P; Overridden method doesnot throw Exception.

Ex ②:-

ⓐ P: public void m1() throws IOException
C: public void m1()

ⓑ P: public void m1()
C: public void m1() throws IOException

ⓒ P: public void m1() throws Exception
C: public void m1() throws IOException

ⓓ P: public void m1() throws IOException
C: public void m1() throws Exception

⑤ P: public void m1() throws IOException

✓ C: public void m1() throws FileNotFoundException, EOFException

⑥ P: public void m1() throws IOException

✗ C: public void m1() throws EOFException, InterruptedException

⑦ P: public void m1() throws IOException

✓ C: public void m1() throws AE, NPE

⑧ P: public void m1()

✓ C: public void m1() throws AE, NPE

Overriding w.r.t Static method :-

→ We Can't override a Static method as non-Static.

Ex:-
```
Class P
{
  Public Static Void m1()
  {
  }
}
Class C extends P
{
  public void m1()
  {
  }
}
```

Static ↓ × ↑ non-Static

C.E:- m1() is Can't override m1() in P; overriden method is Static.

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

→ If both parent & child class method ~~class~~ are static then we wont to get any C.E it seems to be overriding is happen, but it is not overriding. it is "Method Hiding".

Ex:-

```
Class P
{
    Public static void m1()
    {
    }
}
Class C extends P
{
    Public static void m1()
    {
    }
}
```

it is not overriding it is "method hiding"

Method Hiding :-

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

Method hiding	Overriding
1) Both methods should be static	1) Both methods should be non-static
2) Method resolution takes care by compiler based on Reference type.	2) Metho resolution always takes care by JVM based on Runtime object.
3) It is considered as Compiletime Polymorphism or static polymorphism or early Binding	3) It is considered as Runtime Polymorphism or dynamic polymorphism or Late Binding.

Ex:-

```
Class P
{
  public static void m1()
  {
    S.o.pln("parent");
  }
}
Class C extends P
{
  public static void m1()
  {
    So.pln("child");
  }
}
Class Test
{
  p.s.v.m(---)
  {
    P  p = new P();
       p.m1();   --> parent
    C  c = new C();
       c.m1();   --> child
    P  p1 = new C();
       p1.m1();  parent
  }
}
```

method hiding

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

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

→ We can't override a var-arg method with general method. If we are trying to override it will become overloading but not overriding.

→ A var-arg method should be overriden with var-arg method only.

Ex:-

```
Class P
{
  public void m1(int... i)
  {
    S.o.pln("parent");
  }
}
Class C extends P
{
  public void m1(int i)
  {
    S.o.pln("child");
  }
}
Class Test
{
  p.s.v.m(——→)
  {
    P p = new P();
      p.m1(10); // Parent
    C c = new C();
      c.m1(10); // child
    P p1 = new C();
      p1.m1(10); // parent
  }
}
```

overloading but not overriding

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

Overriding w.r.t Variables :-

→ Overriding Concept is not applicable for variables.

→ Variable resolution always takes Care by Compiler based on reference type. Runtime object won't to play any role in variable resolution.

Ex:-

```
Class P
{
  int x = 888;
}
Class C extends P
{
  int x = 999;
}
Class Test
{
  P.S.V.M(———)
  {
    P p = new P();
    S.o.pln(p.x); // 888 ✓
    C c = new C();
    S.o.pln(c.x); //999 ✓
    P p1 = new C();
    S.o.pln(p1.x); 888.
  }
}
```

~~both static~~

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 arguements	must be different (at least order)	must be same (including order).
③ method signature	must be different	must be same.
④ return type	no 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 (or) static polymorphism (or) early binding	runtime polymorphism (or) dynamic polymorphism (or) late binding.

http://javabyanand.blogspot.com

Note:

→ In overloading we have to check only method names (must be same) & arguements (must be diff.) All remaining teams like (return type, throws clause, Access modifiers e.t.c) are not required to check.

→ But in overriding we have to check each & every thing.

Q) Consider the following method declaration in parent class which of the following methods allowed in child class?

P: public void m1(int i) throws IOException

Overriding ✓ ① public void m1(int i)

overloading ✓ ② public void m1() throws Exception

overloading ✓ ③ public static int m1(double d) throws IOException

C.E ✗ ④ public int m1(int i)

C.E ✗ ⑤ public synchronized void m1(int i) throws Exception

overloading ✓ ⑥ public static void m1(int... i) throws Exception

C.E ✗ ⑦ public native abstract void m1().throws Exception.

Polymorphism

poly → many

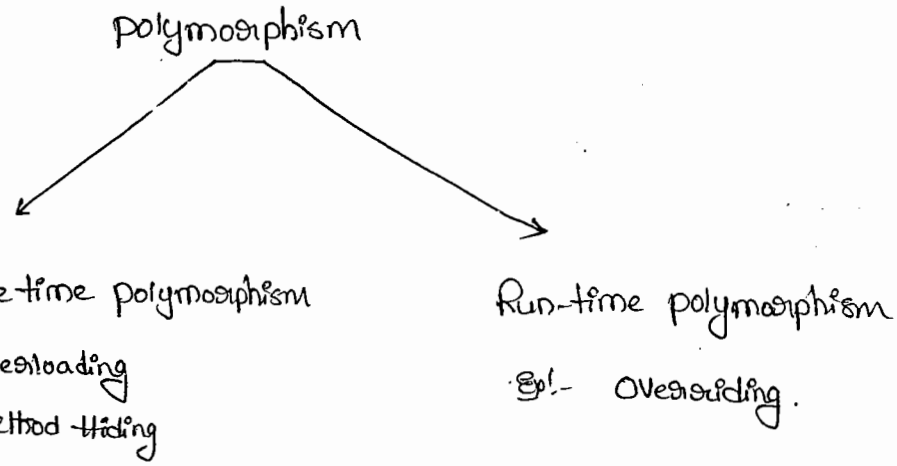
morphs $\xrightarrow{\text{means}}$ forms

i.e polymorphism means many forms

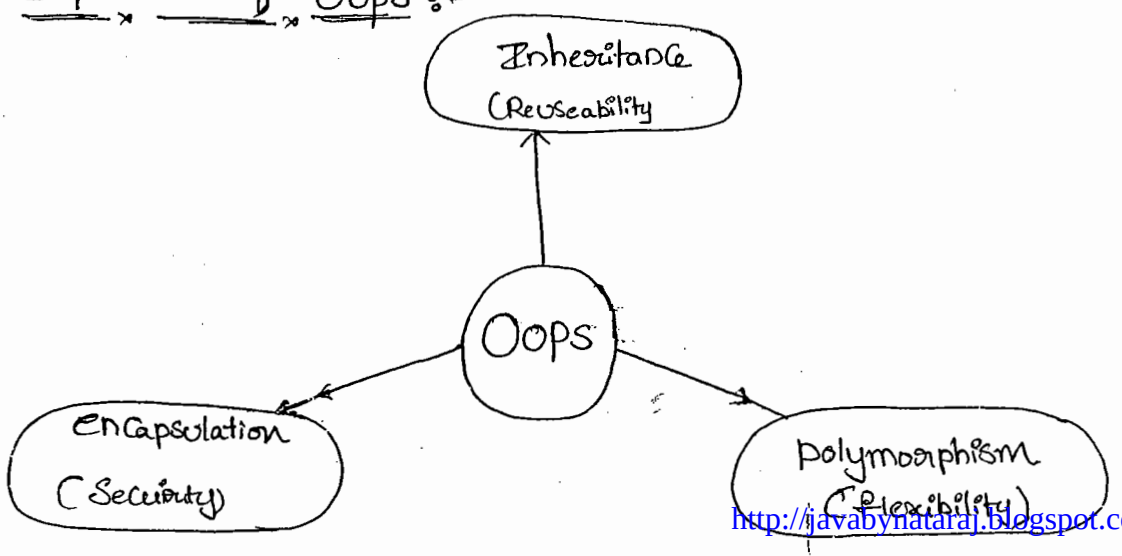
→ we can use same name to represent multiple forms in polymorphism.

Ex:- → In overriding we can have a method with one type of implementation in parent, but different type of implementation in child class.

→ There are 2 types of polymorphism.



3 pillars of Oops :-



funny diffination of polymorphism :-

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

Static Control flow :-

Ex:-

```
Class Base
{
① Static int x=10;                    7
② Static
  {
    m1();                             8
    S.o.pln("FSB");                   10
  }
③ Public static void main(String[] args)
  {
    m1();                             13
    S.o.pln("main method");           15
  }
④ Public static void m1()
  {
    S.o.pln(y);                       9, 14
  }
⑤ Static
  {
    S.o.pln("SSB");                   11
  }
⑥ Static int y = 20;                  12
}
```

x = 0 [RIWO]

y = 0 [RIWO]

x = 10 [R&W]

y = 20 [R&W]

O/P:-

0
FSB
SSB
20
main method

Process:-

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

① identification of static members from Top to bottom. (1 to 6)

② Execution of static variable assignments & static blocks from top to bottom (7 to 12)

③ Execution of main method. (13 to 15)

Read Indirectly write only state (RIWOS)

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

Ex:

```
Class Test
{
  ① Static int x=10;   ③
  ② Static
  {
     S.o.pln(x);
     System.exit(0);
  }
}
```

o/p :- 10

```
Class Test
{
  ① Static
  {
     S.o.pln(x);   ③
  }
  ② Static int x=10;
}
```

C.E:- Illegal-forward reference.

Static block :-

→ At the time of class loading if we want to perform any activity we have to define that activity inside Static block because Static blocks will be Executed at the time of class loading.

→ with in a class we can take any no. of ~~sta~~ 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 ~~mett~~ block to perform this activity at the time of Driver class loading automatically we are not responsible to perform register Explicitly.

```
Ex:  Class Driver
     {
       Static
       {
         Register ~~this~~ Driver with DM
       }
     }
```

Ex(2) :- Advantage:

→ At the time of class loading compulsary we have to ~~to~~ load the Corresponding native libraries hence we can define this step inside Static block.

```
Ex:-  Class Native
      {
        Static
        {
          System.loadLibrary("native Library path");
        }
      }
```

Static Control flow in parents child classes :-

```
Class Base
{
① Static int x=10; ⑫
② Static
  {
    m1(); ⑬
    S.o.pln(" Base SB"); ⑮
  }
③ Public static void main(——)
  {
    m1();
    S.o.pln(" Base main");
  }
④ public static void m1()
  {
    S.o.pln(y); ⑭
  }
⑤ Static int y=20; ⑯
}
Class Derived extends Base
{
⑥ Static int i=100; ⑰
⑦ Static
  {
    m2(); ⑱
    S.o.pln(" DFSB"); ⑳
  }
⑧ Public static void main(——)
  {
    m2(); ㉓
    S.o.p(" Derived main"); ㉕
  }
```

⑨ Public Static void m2()
{
S.o.pln(j); ⑲ ⑳
}

⑩ Static
{
S.o.p("DSSB"); ㉑
}

⑪ Static int j = 200; ㉒
}

> java Derived
O/P:-
0
Base SB
0
DFSB
DSSB
200
Derived main

x = 0 [R I W O]
y = 0 [R I W O]
i = 0 [R I W O]
j = 0 [R I W O]
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

Base.class Derived.class

> java Derived

(1) Identification of Static member from parent to child [1 to 11]

(2) Execution of Static variable assignments & static blocks from parent to child [12 to 22]

*(3) Execution of only child class main method [23 to 25]

(because main() method of parent class is overriding in child class, then, child-class main() method executed)

Process :-

→ whenever we are trying to load child class then automatically parent class will be loaded to make parent class members available to the child class. Hence whenever we are Executing child class the following is the flow with respect to static members step.

(1) Identification of static members from parent to child

(2) Execution of static variable assignments & static blocks from parent to child.

(3) Execution of only child class main method. [If the child class won't contain main method then automatically parent class main() method will be executed].

Note :-

when ever we are loading child class automatically parent class will be loaded. But when ever we are loading parent class child class wont be loaded.

Instance Control flow :-

```
class Parent
{
③ int x=10; ⑨
④ {
    m1();        ⑩
    S.o.p("FIIB"); ⑫
  }
⑤ Parent()
  {
    S.o.pln("Constructor"); ⑮
  }
① Public static void main(String[] args)
  {
② Parent p = new Parent();
    S.o.pln("main");
  }
⑥ Public void m1()
  {
    S.o.pln(y); ⑪
  }
⑦ { ——————→ instance block
    S.o.pln("SIIB"); ⑬
  }
⑧ int y=20; ⑭
}
```

x = 0 [R I W O]
y = 0 [R I W O]
x = 10 [R W]
y = 20 [R W]

O/P :-
0
FIIB
SIIB
Constructor
main

Process :-

→ whenever we are Creating an Object the following Sequence of events will be performed automatically.

(1) Identification of instance member from Top to bottom [1 to 8]

(2) Execution of instance variable assignments & instance blocks from top to botom [9 – 14].

(3) Execution of Constructor [15]

* Note :-

→ Static Control flow is only one time activity and it will be performed at the time of class loading But instance Control flow is not one time activity for every object Creation it will be executed.

Instance Control flow from parent to child :-

```
Class Parent
{
③ int x=10; ⑮
④ {
    m1();  ⑯
    S.o.pln ("parent"); ⑱
  }
⑤ parent()
  {S.o.pln ("parent Constructor"); ⑳
  }
```

```
① public static void main(———)
  {
②   Parent p = new Parent();
    S.o.pln(" ~~child~~ Parent main"); ㉑
  }
⑥ public void m1()
  {
    S.o.pln(y); ⑰
  }
⑦ int y=20; ⑲
  }
  Class child extends Parent
  {
⑩ int i=100; ㉒
  {
⑪   m2(); ㉓
    S.o.pln(" CIIB"); ㉕
  }
⑫ Child()
  {
    S.o.pln(" child Constructor"); ㉘
  }
⑧ Public static void main(———)
  {
⑨   child c = new child();
    S.o.pln(" child main"); ㉙
  }
⑫ public void m2()
  {
    S.o.pln(j); ㉔
  }
```

```
0
parent
Parent Constructor

0
CIIB
CSIIB
Child Constructor
Child main.
```

```
(26) {
        S.o.pln("CSIIB"); (26)
     }
     int j = 200; (27)
}
```

Process :-

→ when even ~~Sequence of~~ we are Creating child class object the following Sequence of ~~execute~~ events will be performed automatically.

(1) Identification of instance member from parent to child.

(2). Execution of instance variable assignments & instance blocks only in parent class.

(3) Execution of parent class Constructor.

(4) Execution of instance variable assignments & instance blocks only in child class.

(5) Execution of child class Constructor.

> java child

> java parent

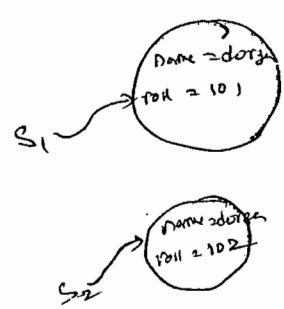
Constructors :-

→ Object Creation is not enough Compulsary we should perform initialization then only that Object is in a position to provide responce properlly.

→ when even we are Creating an Object some peace of the Code will be executed automatically to perform initialization this peace of Code is nothing but Constructor. Here 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 arguement where as instance block Can't take arguements.

→ Similarly we Can't replace instance block with Constructor because a class Can Contain more than one Constructor. if we want to replace instance block with Constructor then in every Constructor we have to write instance blocks Code because at runtime which Constructor will be Called we can't Expect. It results duplicate & Creates maintaince problems.

Ex :-

```
class Test
{
    static int Count = 0;
    Test()
    {
        Count ++;
    }
    Test (int i)
    {
        Count ++;
    }
    p.s.v.m(———)
    {
        Test t1 = new Test();
        Test t2 = new Test(10);
    }
}
```

only once required

if we Create instance

Rules to define Constructors :-

1) The name of the class & name of the Constructor must be matched.

2) Return type Concept is not applicable for Constructor even void also.

By mistake if we declare return type for the Constructor we wont get any Compile-time (or) runtime Errors, because Compiler treats it as method.

Ex:-
```
class Test
{
    void Test()
    {
    }
}
```
→ It is a normal method but not Constructor

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

(3) The only applicable modifiers for Constructors are "public, private, protected, <default> [PPPD]", if we are trying to use any other modifier we will get Compile-time Error saying "modifier xxxx is not allowed here".

→ static/ final/ strictfp ...

Ex:-
```
class Test
{
    final Test()
    {
    }
}
```
C.E:- modifier final is not allowed here.

Singleton classes :-

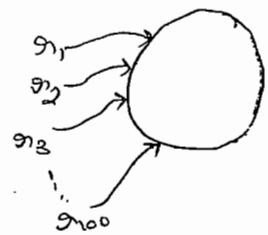
→ for any java class if we are allowed to Create only one Object Such type of class is Called "Singleton class".

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

BusinessDeligate (EJB), ServiceLocator (EJB) ---- e.t.c

→ The main advantage of Singleton is, instead of Creating a Separate Object for every requirement we Can Create a Single object and reuse the Same object for every requirement this approach improves memory utilization & performance of the System.

```
Runtime r1 = Runtime.getRuntime()
Runtime r2 = Runtime.getRuntime()
   ↳ Class   ↳ Static method
   :
Runtime r100 = Runtime.getRuntime()
```

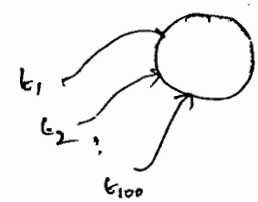


Creation of our own Singleton Class :-

→ we Can Create our own Singleton classes also for this we have to use private Constructor & factory method.

Ex:-
```
class Test
{
  private static Test t;

  private Test()
  {
  }

  public static Test getInstance()
  {
```

```
        if (t == null)
        {
          t = new Test();
        }
        return t;
    }
    public Object clone()
    {
        return this;
    }
}
```




```
Test t1 = Test.getInstance();
Test t2 = Test.getInstance();
   :
Test t100 = Test.getInstance();
Test t101 = Test.clone();
```

factory method:-

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

Ex:-

```
Runtime r = Runtime.getRuntime();          → factory method
DateFormat df = DateFormat.getInstance();  → factory method.
Test t = Test.getInstance();               → factory method
```

→ Similarly we can Create Doubleton, Thrbleton ----- xxxton 12D 96

Classes.

How to Create Doubleton class :-

Ex:-

```
Class Test
{
  private Static Test t1;
  private Static Test t2;
  private Test()
  {
  }
  Public Static Test getInstance()
  {
   if (t1 == null)
   {
     t1 = new Test();
     return t1;
   }
   else
   if (t2 == null)
   {
     t2 = new Test();
     return t2;
   }
  else
   {
    if (math.random() < 0.5)
     return t1;
    else
     return t2;
   }
}
```

Rule :-

Default Constructor :-

→ If we are not writing any Constructor then Compiler will always generate default Constructor.

→ If we are writing atleast one Constructor then Compiler won't generate default Constructor.

→ Hence a class Can Contain either programmer written Constructor (or) Compiler generated Constructor but not both simultaniously.

Prototype of default Constructor :-

1) It is always no arguement Constructor.

2) The access modifier of default Constructor is Same as class modifier but this rule is applicable public & <default>.

3) It Contains only one line, It is a no arguement Call to Super class Constructor.

```
Test()
{
Super();
}
```

Programmers Code	Compiler generated Code

(1)
```
class Test
{
}
```

(1)
```
class Test
{
    Test()
    {
        super();
    }
}
```

(2)
```
public class Test
{
}
```

(2)
```
public class Test
{
    public Test()
    {
        super();
    }
}
```

(3)
```
class Test
{
    void Test()
    {
    }
}
```

→ It is not a constructor
It is a normal method.

(3)
```
class Test
{
    Test()
    {
        super();
    }
    void Test()
    {
    }
}
```

(4)
```
class Test
{
    Test()
    {
    }
}
```

(4)
```
class Test
{
    Test()
    {
        super();
    }
}
```

(5)
```
class Test
{
    Test()
    {
        this(10);
    }
    Test(int i)
    {
    }
}
```

(5)
```
class Test
{
    Test()
    {
        this(10);
    }
    Test(int i)
    {
        super();
    }
}
```

(6)
```
Class Test
{
  Test (int i)
  {
    super();
  }
}
```

5)
```
Class Test
{
  Test (int i)
  {
    super();
  }
}
```

Super & This :-

→ The first Line inside a Constructor should be either super(...) this().

→ If we are not writing any thing Compiler will always places super().

Case (i) :-

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

```
class Test
{
  Test()
  {
    S.o.p("Hi");
    super();   ✗ → C.E:- Call to super must be first Statement in Constructor.
  }
}
```

Case (ii) :-

with in the Constructor we can use either super() or this() but not both simultaniously.

```
Class Test
{
  Test()
  {
    super(); ✓
    this(); ✗ → C.E:- Call to this must be first statement in the Constructor
  }
}
```

Case (iii) :-

→ we can use super & this only inside constructor if we are using any where else we will get compile time error.

Ex:
```
class Test
{
public void m1()
{
    super();   ×  →  C.E:- Call to super must be first statement in the constructor
    S.o.p1n("Hi");
}
}
```

super()
this()

→ must be used only in constructor
→ as the first statement only
→ But not both simultaniously.

[this() :- To call current class constructors
super() :- To call parent class constructors]

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

super() this()	super this
(1) These are constructor calls	1) these are key words to referse 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.m1( )
  {
    S.o.pln( super.hashCode());  X
  }
}
```

↳ CE :- NON-Static variable Super Can't be referenced from a Static Context

Constructor overloading:-

→ A class Can Contain more than one Constructor with Same name but with different arrguements & these Constructors are Considers as overloaded Constructors.

ex:-

```
Class Test
{
  Test(double d)
  {
    this(10);
    S.o.pln(" double-args");
  }
  Test(int i)
  {
    this();
    S.o.pln(" int-args");
  }
  Test()
  {
    S.o.p(" NO-args");
  }
  p.s.v.m(———)
  {
```

Test t_1 = new Test(10.5); → No-args
int-args
double-args

Test t_2 = new Test(10) → No-args
int-args

Test t_3 = new Test() → No-args

** → Inheritance & overriding Concepts are not applicable for Constructors.

** → Every class in java, including abstract class also can contain Constructor. But interfaces can't have the Constructors.

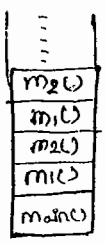
Class Test	abstract Class Test	interface Test
{ Test() { } } ✓	{ Test() { } } ✓	{ Test() ; } ✗

→ <u>Case (i)</u> :-

→ Recursive method Call is always runtime Exception where as recursive Constructor invocation is a Compile time Error.

ex:-

```
Class Test
{
  p.s.v.m1()
  {
    m2();
  }
  p.s.v.m2()
  {
    m1();
  }
  p.s.v.m(——)
  {
    S.o.p("Hello");
    m1();
  }
}
```



R.E.: Stack Over Flow Error

```
Class Test
{
  Test()
  {
    this(10);
  }
  Test(int i)
  {
    this();
  }
  p.s.v.m(——)
  {
    S.o.pln("Hello");
  }
}
```

C.E.: Recursive Constructor invocation.

Case(ii) :-

Ex:-

```
Class P
{
}
Class C extends P
{
}
```

```
class P
{
  P()
  {
  }
}
Class C extends P
{
}
```

```
Class P
{
  P(int i)
  {
  }
}
Class C extends P
{
}
```

C() → ✗

{ super(); }

C.E:-

Can't find Symbol

Symbol : Constructor P()

location : Class P.

Note:-

→ if the parent class contains some constructors then while writing child class we have to take special care about constructors.

→ when ever we are writing any arguement constructor it is highely recommended to write no arguement constructor also.

Case(iii) :-

→ If parent class constructor throws some checked Exception compulsary child class constructor should throw same checked Exception or its parent other wise the code won't compile.

```
class P
{
  P() throws IOException
  {
  }
}
```

```
class C extends P
{
}
```

C.E:- unreported Exception java.io.IOException must be caught or declared to be thrown

http://javabynataraj.blogspot.com/

Ex:-
```
Class P
{
  P() throws IOException
  {
  }
}

Class C extends P
{
    C() throws IOException/Exception
    {
    }
}
```

Q) which of the following is True?

① Every class Contains Constructors ✓

② only Concreate Classes Can Contains 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 Superior this(), ✓
if we are not writing any thing compiler will always place this() ✗

(11) Interface Can Contains Constructor ✗

(12) Both overloading & overriding Concepts are applicable for Constructor ✗.

(13) Inheritance Concept is applicable for Constructor ✗

Type-Casting

Type-Casting:-

→ Parent class reference Can be used to hold child class object

Ex:- Parent p = new child();

→ Similarilly, interface reference Can be used to hold implemented class object.

ex: Runnable r = new Thread();

Syntax:-

```
A b = (C) d;
```

- A → Class|interface
- b → reference variable
- C → Class|interface
- d → Object reference/object

Compiler rule(1):-

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

eg(1) :-

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

eg(2) :-

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

C.E :- inconvertable types
found :- java.lang.String
required : java.lang.SB

Compilerchecking rule 2 :-

→ C must be either same or derived type of A otherwise we will get compiler time error saying "incompatable types"
found : C
required : A

Ex(1) :-

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

Ex(2) :-

```
String s = new String("durga");
StringBuffer sb = (Object) s;
```

C.E :- incompatable types
found : Object
required : SB

Rule 3 :- Runtime checking

→ The underlying object type of 'd' must be either same or derived type of C, otherwise we will get runtime Exception saying "ClassCastException".

Ex:-

① Object o = new String("durga");

SB sb = (SB) o; ✗

Rule ① ✓
② ✓
③ ✗ (R.E):- CCE

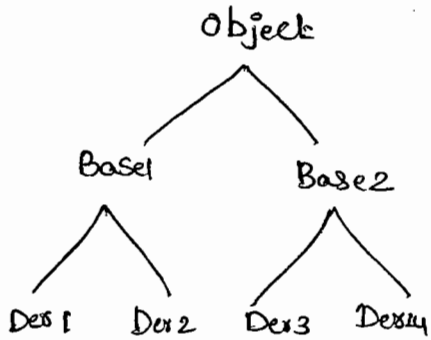
A b = (C) d

② Object o = new String("durga");

String s = (String) o; ✓

Rule ① ✓
② ✓
③ ✓

Ex:-



Ex:- ① Base2 b = new Der4();

② Object o = (Base2) b; ✓

✗ ③ Object o = (Base1) b;

④ Base2 b1 = (Base2) o;

✗ ⑤ Base1 b3 = (Der1)(new Der2());

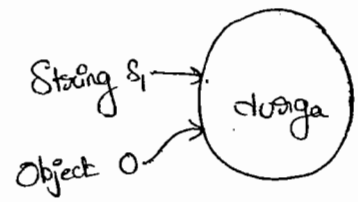
C.E:- inconvertiable types
found : Base2
required : Base1

C.E:- inconvertiable types
found : Der2
required : Der1

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

Ex:-

```
String s1 = new String("durga");
Object o = (object) s1;
S.o.pln(s1 == o); true
```



Ex:-

```
A ---> public void m1()
       {
        S.o.pln("A");
       }
↑
B ---> public void m1()
       {
        S.o.pln("B");
       }
↑
C ---> public void m1()
       {
        S.o.pln("C");
       }
```

```
C c = new C();
c.m1();  o/p C ✓
((B)c).m1();  → C ✓  ---> B b = new C();
                          b.m1();
((A)c).m1();  → C ✓  ---> A a = new C();
                          a.m1();
```

Eg2:-

```
A ---> public static void m1()
       {
        S.o.pln("A");
       }
       ^
       |
B ---> public static void m1()
       {
        S.o.pln("B");
       }
       ^
       |
C ---> public static void m1()
       {
        S.o.pln("C");
       }
```

*) C c = new C();

c.m1(); // C

*) ((B)c).m1(); // B

*) ((A)c).m1(); // A

Eg3:-

```
A ---> int x=777;
       ^
       |
B ---> int x=888;
       ^
       |
C ---> int x=999;
```

C c = new C();

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

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

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

(because the overriding concept is not applicable for variable).

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

Note:-

→ wheather the variable is static or instance variable resolution should be done based on reference type but not based on runtim-object.

Coupling

Coupling :-

→ The degree of dependency b/w the components is called "Coupling"

Ex:-

```
Class A
{
  Static int i = B.j;
}

class B
{
Static int j = C.m1();
}

class C
{
p.s.v.int m1()
{
  return D.k;
}
}

Class D
{
Static int k=10;
}
```

→ the above components are said to be tightly coupled with each other. Tightly coupling is not recommended because it has several serious disadvantages.

(1) with out effecting ~~any~~ remaining component we can modify any component

Hence, enhancement will become difficult.

2. it reduces maintainability.

3. It doesn't promote reusability.

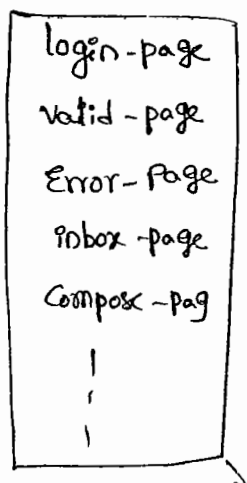
→ Hence it is highly recommended to maintain loosely coupling & dependency b/w the components should be as less as possible.

Cohesion

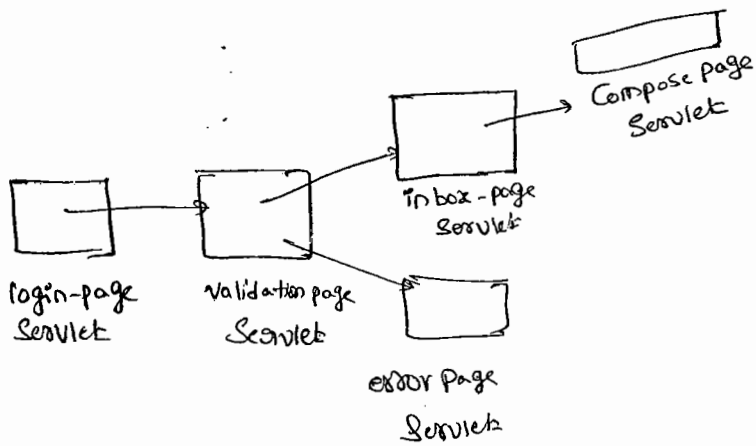
Cohesion :-

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

Ex:-



→ Low-cohesion



→ High-Cohesion

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

(1) without effecting remaining components we can modify any component Hence enhancement will become very easy

(2) It improves maintainability of the application

(3) It promotes reuseability of the code.

Ex:-

→ where even validation is required we can reuse the same validate servlet without rewriting.

Note:-

Loosely coupling & high-cohesion are good programming practices.

=xox=

this :

to use the current class reference

means with out creating multiple objects, only one object is created then

call those values from the current class.

Kondalu_7@yahoo.co.in

మో, పే

① Collection framework (1 — 43)

Collection (6 — 24)

Map (25 - 37)

Collections class (38 — 40)

Arrays class (41 - 43)

② Generics (44 - 52)

③ Multithreading (53 - 76)

Synchronization (67 - 76)

④

1 Re

④ Regular Expressions (77 - 82)

⑤ Enumeration (83 - 89)

⑥ IBN (90 - 95)

⑦ Development (96 - 101

Durgasoft SCJP Notes

Part-2

12/03/11

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. Hence, to use arrays concept compulsary we should know the size in advance, which may not possible always.

(2) 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[];  ✗  CE:- Incompatiable 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 readymed method support is not available for every requirement. Compulasary 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 datastructure. Hence ready-made 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 Concept is not recommended to use	2) memory point of view Collections 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 ready-made method support is not available	5) underlying D.S is available for every Collection class. Hence ready-made 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 :-

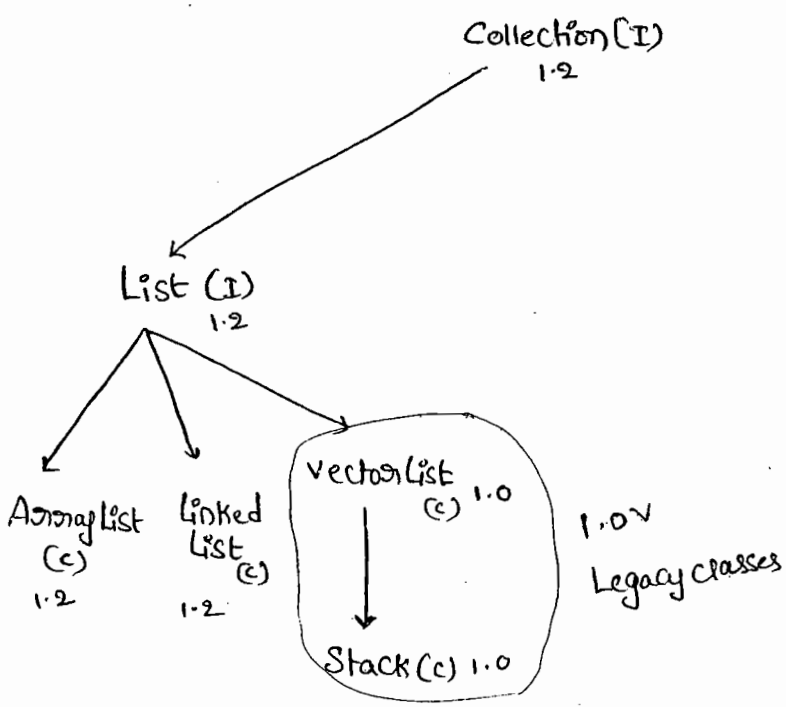
→ Collection is an interface, Can be used to represent a group of individual object as a Single Entity. where as,

→ Collections is an Utilityclass, present in java.util package, to define Several utility methods for Collections.

3) 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.

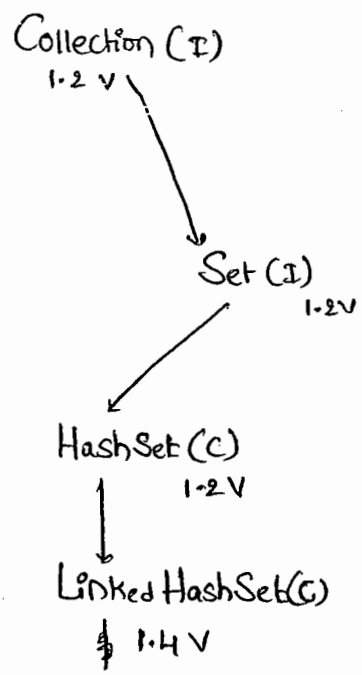


→ Vector & Stack Classes are re Engineered in 1.2 version to fit into Collection frame Work.

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):-

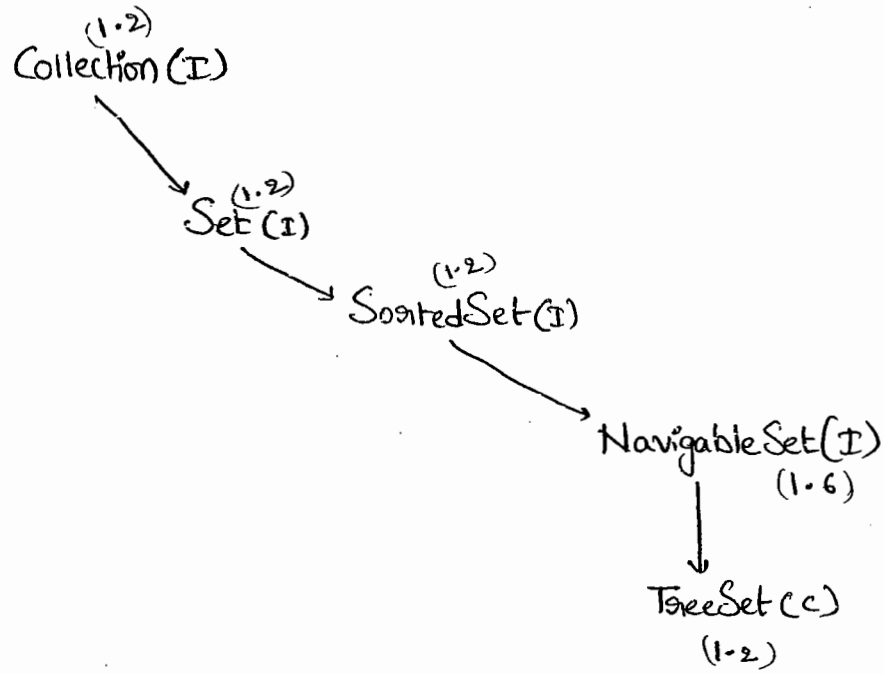
→ It is the child interface of Set.

→ 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.

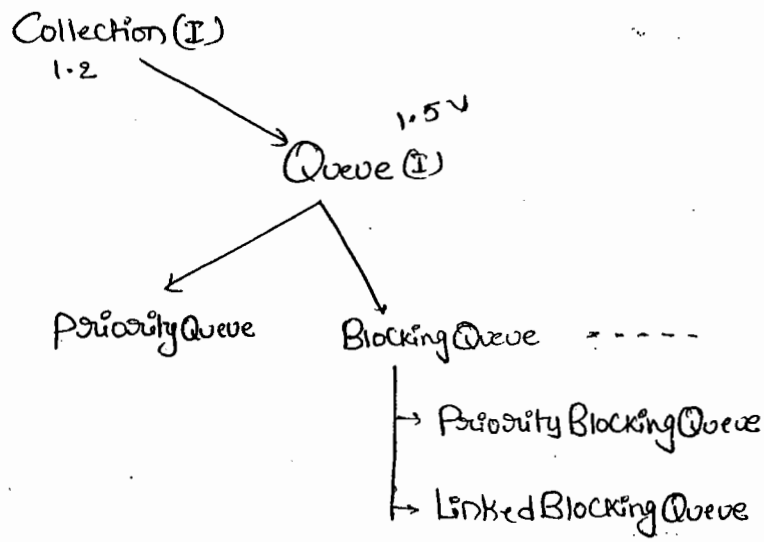
→ It is introduced in 1.6 version.



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) ment for representing a group of individual objects.

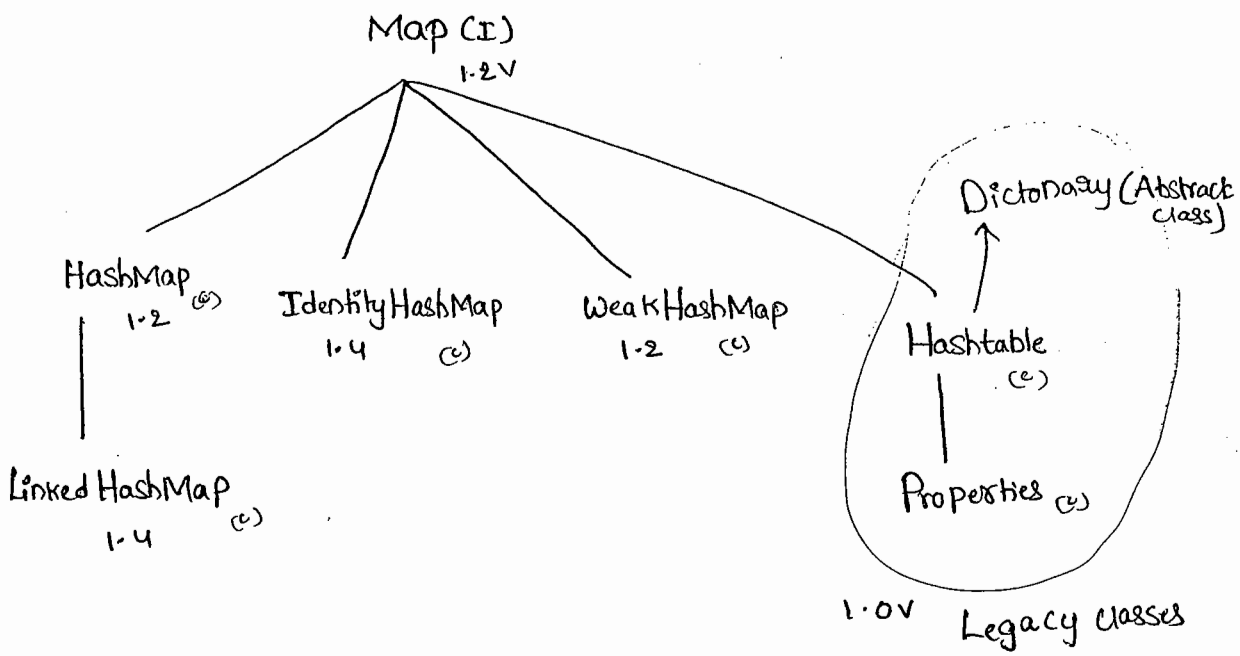
→ If we want to represent a group of objects as key-value pairs then we should go for 'Map'.

(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):-

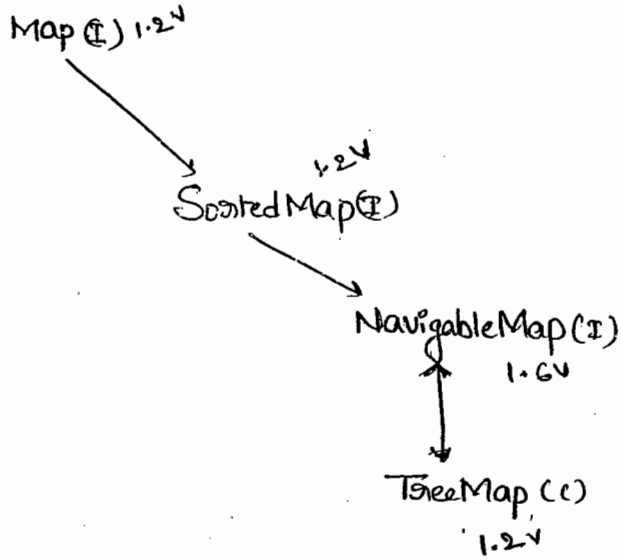
→ If we want to represent a group of objects as Key-value pairs according to Some Sorting Order. Then we Should go for SortedMap.

→ Sorting should be done Only based on Keys. but not based on values.

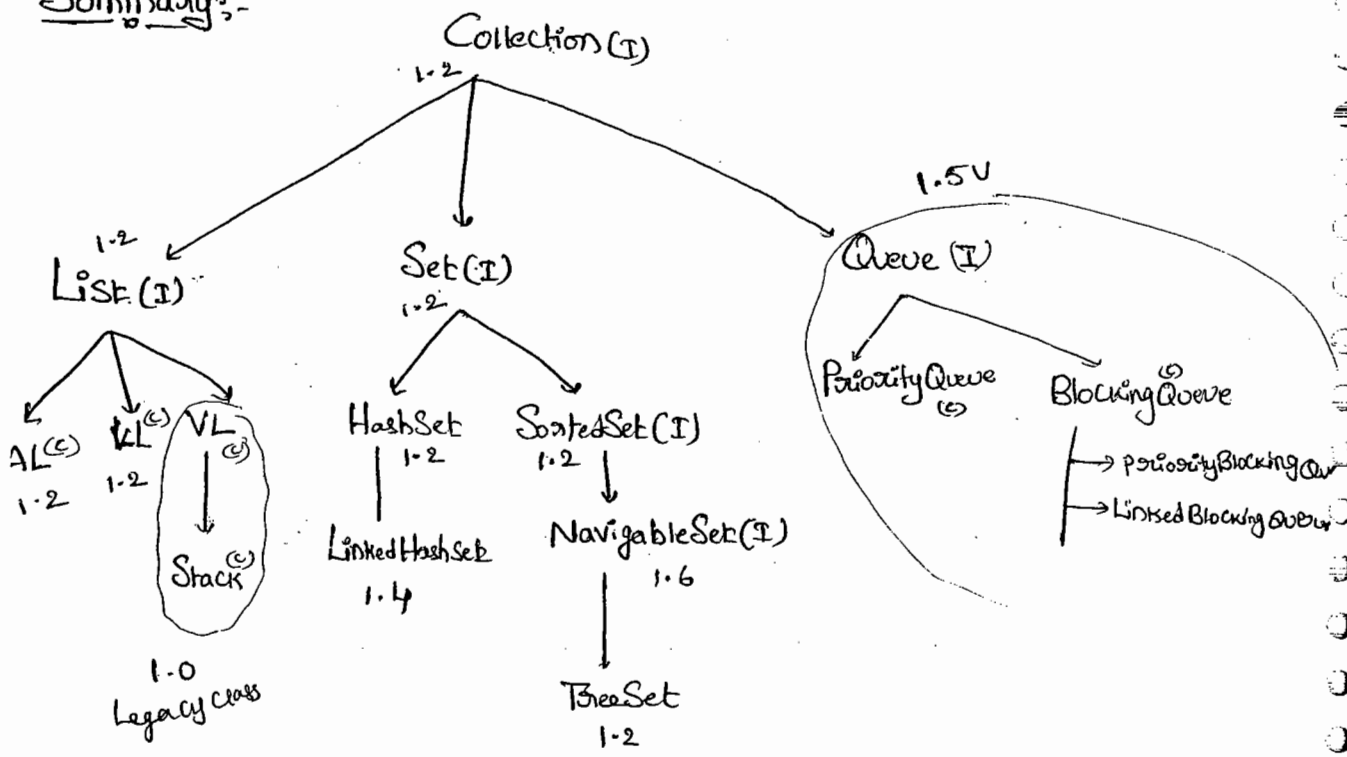
→ SortedMap is child Interface of Map.

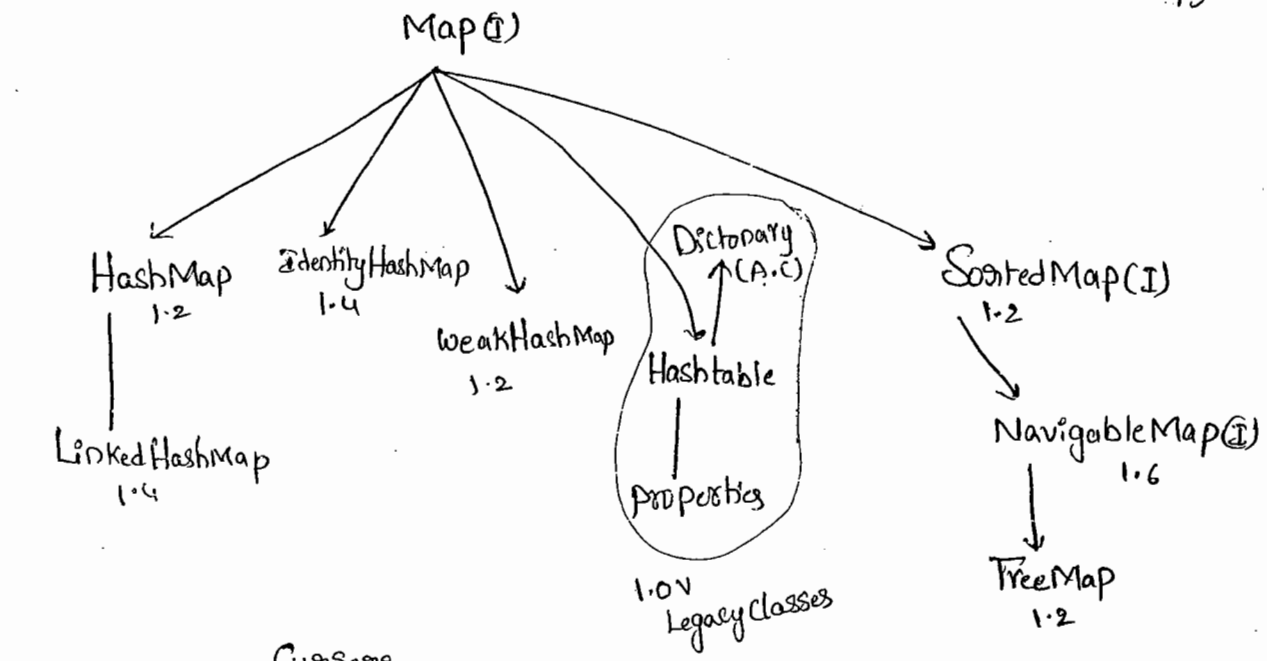
(9) NavigableMap (I):-

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



Summary:-





Arrays
Collections

Cursors
Enumaration
Iterator
ListIterator

Comparable
Comparator

→ In the Collection frame work the following are Legacy characters

(1) Enumaration (I)
(2) Dictionary (Ab 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 differenciate duplicate objects by using Index. Hence Index place 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 set(int index, Object new)
   old
6. int indexOf(Object o)
7. int lastIndexOf(Object o)
8. ListIterator listIterator()

It contains 4 classes :-

(i) ArrayList (c) :- 1.2

(ii) LinkedList (c) :- 1.2

(iii) VectorList (c) :- 1.0

(iv) Stack (c) :- 1.0

(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 objects

i.e, This Constructor is for dancing b/w Collection objects

Ep:-

```
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);
    S.o.pln(a);      [A, 10, A, null]
                      0   1   2   3
    a.remove(2);
    S.o.pln(a);      [A, 10, null]
    a.add(2,"M");    [A, 10, M, null]
    a.add("N");      [A, 10, M, null, N]
    S.o.pln(a);      [A, 10, M, null, N]
    S.o.pln(a.size());//5
    a.clear();// []
    a.addAll(a);// [A, 10, M, null, N, A, 10, M, null, N]
  }
}
```

Note:-

* In Every Collection class toString() is overridden to return its Content directly in the following formatt.

  [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 RandomAccess Interface, So that any random element we can access with same speed. Hence, if our frequent operation is retrivable 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).

Q: differences b/w ArrayList & Vector?

ArrayList	Vector
① No method is Synchronized	① Every method is Synchronized
② multiple threads can access ArrayList simultaneously, hence ArrayList object is not threadsafe	② At any point only one thread is allowed to operate on vector object at a time. Hence vector 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 1.2 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)
```

eg:-

```
ArrayList l = new ArrayList();
List l1 = Collections.SynchronizedList(l)
```

l1 → Synchronized

l → 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.

i) 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-interfaces.

→ Best Suitable if our frequent operation insertion or deletion in the middle.

→ Worst choice if our frequent operation is retrival.

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
{
  p.s.v.m(String[] args)
  {
    LinkedList l = new LinkedList();
    l.add("durga");
    l.add(30);
    l.add(null);
[durga, 30, null, durga]  l.add("durga");
[S/w, 30, null, durga]  l.set(0, "Software");
[venky, S/w, 30, null, durga]  l.add(0, "venky");
[venky, S/w, 30, null]  l.removeLast();
[ccc, venky, S/w, 30, null]  l.addFirst("ccc");
    S.o.pln(l);
  }
}
```

ccc venky ~~durga~~ software 30 null ~~durga~~

[ccc, venkey, software, 30, null]

ii) Vector(c) :-

→ The underlying datastructure is Resizeable 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 Retrival & worrest choice if our frequent operation is insertion or deletion in the middle.

→ Every method in vector is Synchronized. Hence vector object is ThreadSafe.

Constructors:-

(i) Vector v = new Vector();

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

→ One vector reaches it's max. Capacity a new vector object will be Created with double Capacity.

New Capacity = 2* 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 :- (10)

→ 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 retrive elements

① get(int index) ⟶ L

② elementAt(int index) ⟶ V

③ firstElement(); ⟶ V

④ LastElement(); ⟶ V

→ Other methods

① int Size();

② int Capacity();

* ③ Enumeration elements();

Eg:-

```
import java.util.*;
class VDemo1
{
  p.s.v.m(String[] args)
  {
    Vector v = new Vector();
    S.o.pln(v.capacity());
    for(int i=1 ; i<=10 ; i++)
    {
      v.addElement(i);
    }
    S.o.pln(v.capacity());
    v.addElement("A");
    S.o.pln(v.capacity());
    S.o.pln(v);
  }
}
```

O/P:-

```
10
10
20
[1, 2, 3, 4, 5, 6, ----10, A]
```

v.size() // 11 ← no. of objects

v.removeElement(9) // [1, 2, 3, 4, 5, 6, 7, 8, 10, A] ← object

v.removeElementAt(3) // [1, 2, 3, 5, 6, 7, 8, 10, A]

v.removeAllElements() // []

(iv) 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.

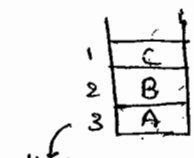
(iv) boolean Empty();

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.

Exp.



S.search("A"); 3
S.search("C"); 1
S.search("Z"); -1

Exp.

```
import java.util.*;
class StackDemo
{
  p.s.v.m(String[] args)
  {
    Stack s = new Stack();
    S.push("A");
    S.push("B");
    S.push("C");
    S.o.pln(S); / [A B C]
    S.o.pln(S.Search("A")); 3
    S.o.pln(S.Search("Z")); -1
  }
}
```

S.pop(); [A B C] C A B

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 (in 1.0 Ver)

→ 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();
```

eg:-

```
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
{
 p.s.v.m(String[] args)
 {
   Vector v = new Vector();
   for(int i=0 ; i<=10 ; i++)
   {
     v.addElement(i);
   }
   S.o.pln(v);   [0, 1, 2, 3 ----10]
   Enumeration e = v.elements();
   while(e.hasMoreElements())
   {
    Integer I = (Integer)e.nextElement();
    if(I%2 ==0)
    S.o.pln(I);
   }
   S.o.pln(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 itr = C.iterator()
```

Any collection object

→ Iterator Interface defines the following 3 methods.

(i) public boolean hasNext();

(ii) public Object next();

(iii) 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);
   S.o.pln(h.add("Z")); // false
   S.o.pln(h); // [null, D, B, C, 10, Z]
 }
}
```

o/p:- false
[null, D, B, C, 10, Z]

Note:- Insertion order is not preserved

(ii) LinkedHashSet (c):-

→ LinkedHashSet is the child class of HashSet.

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

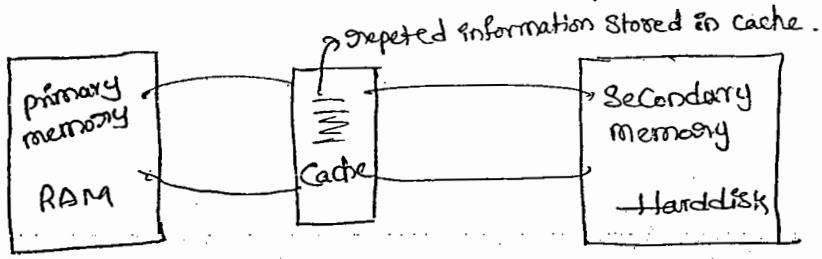
HashSet	LinkedHashSet
(i) The underlying D.S is Hashtable	i) The underlying D.S is a Combination of Hashtable & Linked List
(ii) Insertion order is not preserved	ii) Insertion order is preserved.
(iii) Introduced in 1.2V	iii) Introduced in 1.4V

→ 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) :-

→ It 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)

→ returns the SortedSet whose elements are greaterthan or equal to obj ($\geq$)

(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 assending order

→ The default natural sorting order for characters & strings are is alphabetical order (dictionary based order).

TreeSet (c) :-

→ The underlying datastructure 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 first obj~~ 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)
  {
```

```
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
    S.o.pln(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
{
  p.s.v.m(String[] args)
  {
    TreeSet t = new TreeSet();
      t.add(new StringBuffer("A"));
      t.add(new StringBuffer("Z"));
      t.add(new StringBuffer("L"));
      t.add(new StringBuffer("B"));   o/p:- CCE
    S.o.pln(t);
```

↳ If we are depending on default natural Sorting order Compolsary 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 where as StringBuffer doesn't implements Comparable Interface. Hence, In the above Example we got ClassCastException.

Comparable Interface :-

→ This Interface present in java.lang package & Contains only one method i.e., "compareTo()".

```
public int compareTo(Object obj)
```

```
obj1.compareTo(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)

eg:-
```
import java.util.*;
class Test
{
  p.s.v.m(String[] args)
  {
    S.o.pln("A".compareTo("z")); // -ve   -25
    S.o.pln("z".compareTo("K")); // +ve   15
    S.o.pln("A".compareTo("A")); // 0     0
  }
}
```

13/03/11

→ When were are depending on default natural Sorting order internally JVM Calls ~~Com~~ compareTo().

→ Based on the return-type JVM identifies the Location of the element in Sorting order.

obj1 . compareTo (obj2)

which object were trying to add

already existing object in TreeSet.

→ 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".compareTo("z"); -ve
t.add("D");   ⟹ "D".compareTo("K"); -ve
t.add("M");   ⟹ "M".compareTo("D") ⟹ +ve
t.add("D");      "M".compareTo("K") ⟹ +ve
                 "M".compareTo("z"); → -ve
// t.add(null);   "D".compareTo("D") ⟹ 0
S.o.pln(t);
}
[0, K, M, z]     ~~ClassCastException: NPE~~
                 null.compareTo("D") ⟹ RE ⟹ NPE
```

→ If we are not satisfied with default natural sorting order (or) if the default natural sorting order is not already available. Then we can define our own Customized sorting by using Comparator

* Comparable ment for default natural sorting order.
* Comparator ment 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

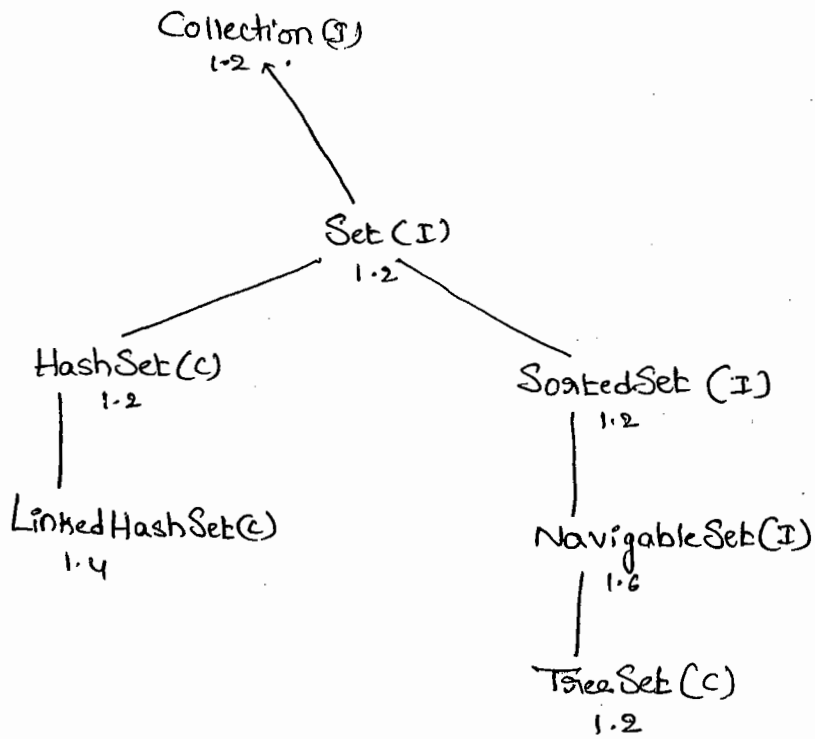
② public boolean equals(Object obj)

→ when ever we are implementing Comparator Interface compulsary we should provide implementation for compare(), 2nd method equals() implentation is optional, because it is already available for our class from Object class through Inheritance.

(2) Set (I) :-

→ 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 doesnot 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.

→ Insertion order is not preserved & all objects are inserted according to hashcode of the objects.

http://javabynataraj.blogspot.com

→ 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 ~~diffault~~ 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 (filling) the Specified ratio then only a new HashSet object will be Created that particular ratio is called fillratio & 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);
    }
    S.o.pln(l);   [0,1,2,3,- - - -10]
    Iterator itr = l.iterator();
    while(itr.hasNext())
    {
      Integer I = (Integer)itr.next();
      if(I%2 ==0)
      {
        S.o.pln(I);      0
      }                  2
                         4
                         6
      else               8
                         10
      {
        itr.remove();
      }
    }
    S.o.pln(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 performaning 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.

i) ListIterator :-

→ 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() of List Interface.

```
ListIterator litr = l.listIterator();
```

→ any List object
→ small Letter

→ 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("chiru");
     l.add("nag");
     S.o.pln(l);      [balakrishna, venky, chiru, nag]
     ~~LinkedList~~
     ListIterator ltr = l.listIterator();
     while (ltr.hasNext())
     {
      String s = (String)ltr.next();
      if (s.equals("venki"))
      {
        ltr.remove();
      }
      if (s.equals("chiru"))
       {
        ltr.set("charan");
       }
      if (s.equals("nag"))
       {
        ltr.add("chittu")
       }
      S.o.pln(l);          [Balakrisha, charan, nag, chittu]
     }
 }
```

Note:-

→ among 3 cursors ListIterator is the most powerfull cursor, But it is applicable only for List objects.

Comparision table of 3-cursors :-

Property	Enumeration (1.0V)	Iterator (1.2V)	ListIterator (1.2V)
① Is it legacy	yes	No	No
2) It is applicable only for	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)
  {
    ~~Tree 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
              -> Compare(5, 0) -> +ve
              -> Compare(5,15) -> -ve
    t.add(10); -> Compare(10, 20) +ve
               -> Compare(10, 0) +ve
               -> Compare(10, 15) +ve
               -> Compare(10, 5) -ve
    S.o.println(t);
    [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 -1000;
    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 ment 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 ment 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]
 }
}
```

(*) w.a.p to insert String objects into the TreeSet where the Sorting order is reverse of alphabetical order.

(A):

```
import java.util.*;

class TreeSetDemo2
{
  Public Static . V. m(String[] args)
  {
    TreeSet t = new TreeSet(new myComparator());
      t.add("A");
      t.add("Z");
      t.add("K");
      t.add("B");
      t.add("a");
    S.o.pln(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:- ~~In object class Comparee method doesn't Contain Strings only Contain object type so objects Can be Convert into Strings by using typecasting~~

Note:-

→ In objects & StringBuffer there is no CompareTo(), So we Can Convert into Strings.

* 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
{
  p.s.v.m(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");
    S.o.pln(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 > l2)
      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
{
  P.S.V.M(String[] args)
  {
    TreeSet t = new TreeSet(new MyComparator());
      t.add(new StringBuffer("A"));
      t.add(new StringBuffer("Z"));
      t.add(new StringBuffer("K"));
                              ("L"));
      S.o.pln(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]

Note :-

So SB can be convert into String

→ In StringBuffer there is no CompareTo method

→ If we are depending on default natural Sorting order Compulsary 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 Compulsary we should define Sorting by using Comparator object only.

Ex:- StringBuffer.

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()
  {
    returns "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 CompCompDemo
{
  p s.v.m(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);
t2.add(e2);
t3.add(e3);
t4.add(e4);
t1.add(e5);
S.o.pln(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.pln(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);
```

1), W.a.p 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.,&

* W.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.

* Comparision b/w Comparable & Comparator :-

Comparable	Comparator
1) We can use Comparable to define default natural Sorting order.	1) We can use Comparator to define Customized Sorting order.
2) This interface present in Java.lang package.	2) This interface present in java.util package.
3) defines only one method i.e Compareto()	3) defines Two methods (i) Compare() (ii) equals()
4) All wrapper classes & String Class implements Comparable interface	4) No predefined class implements Comparator Interface.

Comparision table for Set implemented classes.?

Property	HashSet	LinkedHashSet	TreeSet
1) Underlying D.S	Hash table	Hashtable + Linked List	Balanced Tree
2) insertion order	not preserved	preserved	not preserved
3) Sorting order	N.A	N.A	preserved
4) Heterogeneous Objects	allowed	allowed	Not allowed
5) Duplicate objects	not allowed	not allowed	not allowed
6) null acceptance	allowed (1)	allowed (1)	for the empty TreeSet add the first element null insertion is possible. in all other cases we will get NPE

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:-

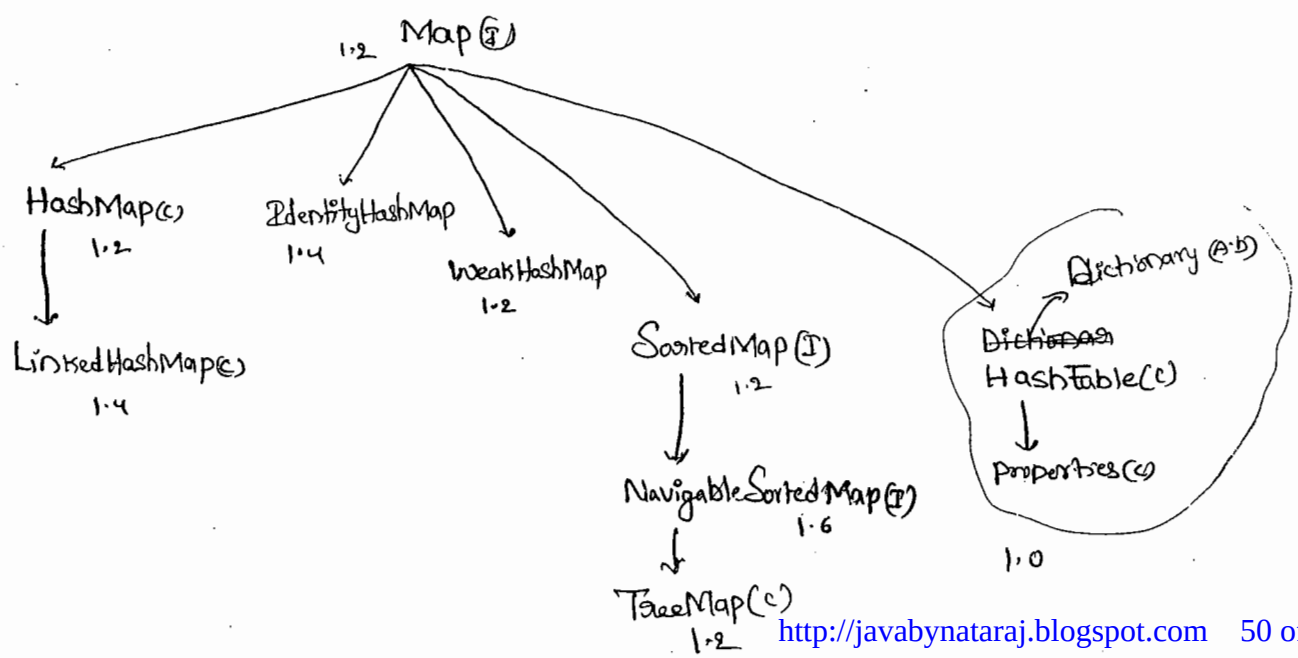
Rollno	name
101	durga
102	Srinu
103	Ravi
104	Sambu
105	Sundar

key, value, entry

→ There is no relationship b/w Collection & Map.

→ * Collection meant for a group of individual objects where as
* 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()

① Set keySet();
② Collection values();
③ 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();
  }
}
```

(1) HashMap :-

→ The underlying datastructure 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 Operates Simultaneously & Hence HashMap object is not Thread Safe	② At a time only one Thread is allowed to Operate on HashTable Objects. 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 NPException
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 EmptyHashMap 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
{
 p.s.v.m(String[] args)
 {
   HashMap m = new HashMap();
    m.put("chiranjeevi", 700);
    m.put("balaiah", 800);
    m.put("venkatesh", 1000);
    m.put("nagarjuna", 500);
    S.o.pln(m);   {venkatesh =1000, balaiah =800, chiranjeevi=700,
                                                  nagarjuna = 500}
    S.o.pln(m.put("chiranjeevi", 1000));   700
    Set s = m.keySet();
    S.o.pln(s);   [venkatesh, balaiah, chiranjeevi, nagarjuna]
    Collection c = m.values();
    S.o.pln(c);   [1000, 800, 1000, 500].

    Set S1 = m.entrySet();
    Iterator its = S1.iterator();
```

```
while (its.hasNext())
{
    Map.Entry m1 = (Map.Entry) its.next();
    S.o.pln(m1.getKey() + "----" + m1.getValues());
    if (m1.getKey().equals("nagarjuna"))
        m1.setValue(10000);
}
S.o.pln(m);
}
}
```

nagarjuna 500
venkatesh 1000
balaiah 800
chiranjeevi 1000

{nagarjuna =10000, venkatesh =1000, balaiah =800, chiranjeevi=1000}

ii) Linked HashMap :-

→ It is the child class of HashMap.

→ It is Exactly Same as HashMap Except the following differences

HashMap	Linked HashMap
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 Linked HashMap, The following is the O/P.

{chiranjeevi =700, balaiah =800, venkatesh=1000, nagarjuna =500}

i.e insertion order is preserved

Note:-

→ The main application area of LinkedHashSet & LinkedHashMap are cache applications implementation where duplication is not allowed & insertion order must be preserved.

(iii) IdentityHashMap :-

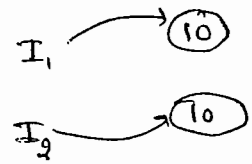
→ It is Exactly Same HashMap Except the following difference.

→ In the Case of HashMap to identify duplicate Keys JVM always uses .equals(), which is mostly ment for Content Comparision.

→ If we want to use == operator instead of .equals() to identify duplicate keys we have to use IdentityHashMap. (== operator always ment for reference Comparision).

Eg:-

```
HashMap m = new HashMap();
Integer i1 = new Integer(10);
Integer i2 = new Integer(10);
m.put(i1, "pavan");
m.put(i2, "Kalyan");
S.o.pln(m); {10=Kalyan}
```



.equals() → Content

== → reference

I1 == I2 → false

I1.equals(I2) → True

→ 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 eleigible for g.c eventhough 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 eventhough object associated with WeakHashMap, it is eligible for g.c, if it doesnot 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.pln(m);        {temp = durga}
    t = null;
    System.gc();
    Thread.sleep(5000);
    S.o.pln(m);
  }
}                      {temp = durga}
```

```
Class Temp
{
  Public String toString()
  {
    return "temp";
  }
  public void finalize()
  {
    System.out.println("finalize method called");
  }
}
```

O/P: {temp = durga}
{temp = durga}

→ If we replace HashMap with WeakHashMap then the o/p is

{temp = durga}
finalize method called
{}

(ii) SortedMap (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 key1);
4. SortedMap tailMap(Object key1);
5. SortedMap subMap(Object key1, Object key2)
6. Comparator comparator();

(iii) TreeMap (C) :-

→ The underly 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 (CCE).

→ 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 any where 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(SortedMap m)

Eg:-

```
import java.util.*;
class TreeMapDemo3
{
  p.s.v.m(String[] args)
  {
    TreeMap m = new TreeMap();
    m.put(100, "zzz");
    m.put(103, "yyy");
    m.put(101, "xxx");
    m.put(104, 106);
    m.put(107, null);
    //m.put("FFFF", "xxx"); //CCE
    // m.put(null, "xxx"); //NPE
    S.o.p1n(m);  {100=zzz, 101=xxx, 103=yyy, 104=106, 107=null}
  }
}
```

O/P:-

{100=zzz, 101=xxx, 103=yyy, 104=106, 107=null}.

eg:-

```
import java.util.*;

Class TreeMapDemo
{
  p.s.v.m(String[] args)
  {
    TreeMap t = new TreeMap(new MyComparator());
     t.put("xxx", 10);
     t.put("AAA",20);
     t.put("zzz", 30);
     t.put("LLL", 40);
     S.o.pln(t);
  }
}

Class MyComparator implements Comparator
{
  public int compare(Object obj1, object obj2)
  {
     String S1 = Obj1.toString();
     String S2 = obj2.toString();
     return S2.compareTo(S1);
  }
}
```

o/p:- {zzz = 30, xxx = 10, LLL=40, AAA =20}

Hashtable(c):

→ 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("durga", null); //NPE
    System.out.println(h);
  }
}
```

10	
9	
8	
7	
6	6=C
5	5=A, 16=F
4	15=D
3	
2	2=B
1	23=E
0	

{6=C, 16=F, 5=A, 15=D, 2=B, 23=E}

from top to bottom & Right to Left

```
Class Temp
{
  int i;
  Temp(int i)
  {
    this.i = i;
  }
  public int hashCode()
  {
    return i;
  }
  Public String toString()
  {
    return i+"";
  }
}
```

(ii) Properties (C):-

→ It is the child class of Hashtable

→ In our program if any thing which changes frequently (like database usernames, passwords, URL) never recommended to hardcode the value in the java program. Because for every change, we have recompile, rebuild, redeploye the application & sometime even server restart also required. which creates a big business impact to the client.

→ we have to configuare those variables (~~properties~~ inside properties files & we have to read those values from javacode.

→ the main advantage of this approach is, If any change in the properties file just redeployement 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(String PropertyName)

→ returns the value associated with specified property.

(ii) ~~String~~ setProperty(String pname, String pvalue);

→ to set a new property.

(iii) ~~String~~ Enumeration propertyNames();

* (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
{
  P.S.V.M(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");
    S.o.pln(s);
    p.setProperty("nag", "999999");
    FileOutputStream fos = new FileOutputStream("abc.properties");
    p.store(fos, "Updated by durga for SCJP Demo Class");
  }
}
```

User = scott venki = 8888 Pwd = tiger

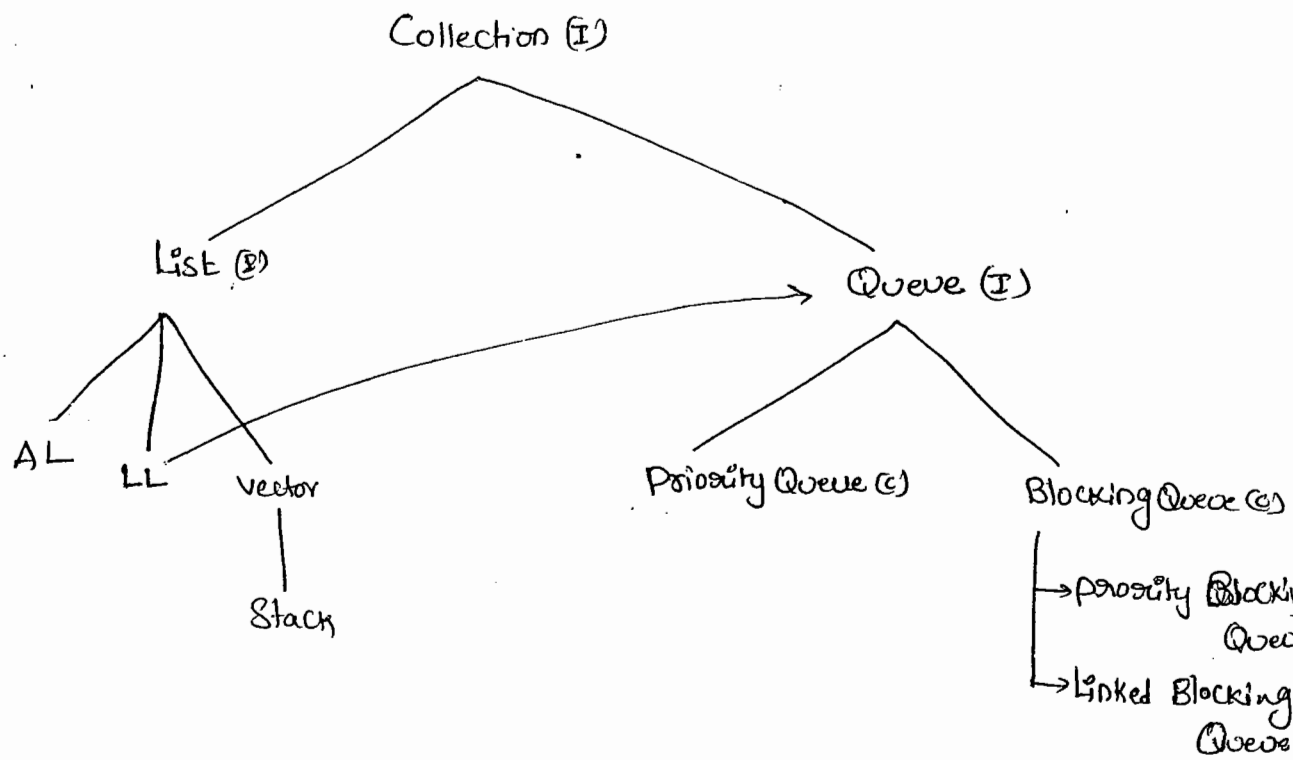
abc.properties

1.5 Version Enhancement :-

Queue (I) :-

→ It is the child Interface of Collection.

→ if we want to Represent a group of individual objects prior to processsoring 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 NoSuchElementExcept

PriorityQueue (c) :-

→ 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 Compulsary 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) PriorityQueue q = new PriorityQueue();

→ Creates an Empty PriorityQueue with default initialCapacity "11" & Priority order is default natural Sorting order.

(ii) PriorityQueue q = new PriorityQueue(int initialCapacity);

(iii) PriorityQueue q = new PriorityQueue(int initialCapacity, Comparator c);

(iv) PriorityQueue q = new PriorityQueue(Collection c);

(v) PriorityQueue q = new PriorityQueue(SortedSet s)

Eg:-

```
import java.util.*;
Class PriorityQueueDemo
{
  p.s.v.m(String[] args)
  {
    PriorityQueue q = new PriorityQueue();
    S.o.pln(q.peek()); //null
    // S.o.pln(q.element()); //NSE  NoSuchElementException
    for(int i=0 ; i<=10 ; i++)
    {
      q.offer(i);
    }
    S.o.pln(q); // [0, 1, 2, 3, 4, 5, ----10]
    S.o.pln(q.poll()); // 0
    S.o.pln(q); // [1, 2, 3, 4, 5 ---10]
  }
}
```

eg 2:-

```
import java.util.*;
Class PriorityQueueDemo2
{
  p.s.v.m (String[] args)
  {
```

```
PriorityQueue q = new PriorityQueue(15, new MyComparator());
    q.offer("A");
    q.offer("Z");
    q.offer("L");
    q.offer("B");
    S.o.pln(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);
  }
}
```

O/P:. [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 >= e

(ii) higher(e) :-

→ returns the lowest element which is >e.

(iii) floor(e) :-

→ returns highest element which is <= 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
{
 p.s.v.m(String[] args)
 {
   TreeSet<Integer> t = new TreeSet<Integer>();
    t.add(1000);
    t.add(2000);
    t.add(3000);
    t.add(4000);
    t.add(5000);
    S.o.pln(t); [1000, 2000,
    S.o.pln(t.ceiling(2000)); 2000
    S.o.pln(t.higher(2000)); 3000
    S.o.pln(t.floor(3000)); 3000
    S.o.pln(t.lower(3000)); 2000
    S.o.pln(t.pollFirst()); 1000
    S.o.pln(t.pollLast()); 5000
    S.o.pln(t.descendingSet()); [4000, 3000, 2000]
    S.o.pln(t); [2000, 3000, 4000]
 }
}
```

(ii) NavigableMap (I):-

→ It is the child interface of SortedMap to define several method 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)
  {
    TreeMap<String, String> t = new TreeMap<String, String>();
     t.put ("b", "banana");
     t.put ("c", "cat");
     t.put ("a", "apple");
     t.put ("d", "dog");
     t.put ("g", gun");
     S.o.pln(t); { a=apple, b=banana, c=cat, d=dog, g=gun}
```

```
S.o.pln(t.ceilingKey("c"));    c
S.o.pln(t.higherKey("e"));    g
S.o.pln(t.floorKey("e"));    d
S.o.pln(t.lowerKey("e"));    d
S.o.pln(t.pollFirstEntry());  a = apple
S.o.pln(t.pollLastEntry());   g = gun
S.o.pln(t.descendingMap());  {d = dog , c = cat , b = banana}
S.o.pln(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 comparabry elements should be Homogeonus & 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 BinarySearch algorithm.

→ Before Calling binarySearch() method Compulsary 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.pln(l);      [Z, A, M, K, a]
Collections.sort(l);
S.o.pln(l);      [A | K | M | Z | a]
S.o.pln(Collections.binarySearch(l,"z"));   3
S.o.pln(Collections.binarySearch(l,"j"));   -2
}
}
```

-1	-2	-3	-4	-5	-6
A	K	M	Z	a	
0	1	2	3	4	

Ex2:-

```
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.pln(l);      [15 | 0 | 20 | 10 | 5]
    Collections.sort(l, new MyComparator());
    S.o.pln(l);      [20 | 15 | 10 | 5 | 0]
    S.o.pln(Collections.binarySearch(1,10,new MyComparator()));  //2
    S.o.pln(Collections.binarySearch(1,13,new MyComparator()));  //-3
    S.o.pln(Collections.binarySearch(1,17));  // -6 unpredictable
  }
}
```

-1	-2	-3	-4	-5	-6
20	15	10	5	0	
0	1	2	3	4	

because it is not passing Comparator()

```
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 unsuccessfull 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.pln(l);
        Collections.reverse(l);
        S.o.pln(l);
    }
}
```

15	0	20	10	5

5	10	20	0	15

reverse() Vs reverseOrder() :-

→ We can use reverse() method to reverse the elements of a List and this method contain List arguement

→ Collections class defines reverse order method also to return Comparator object for reversing original sorting order

```
Comparator c1 = Collections.reverseOrder(Comparator c)
```

c1 → decending order

c → asending order

→ reverseOrder() method ~~can take~~ contains Comparator arguement whereas reverse() contains List arguements.

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);
      S.o.pln(l);
      Collections.reverse(l);
      S.o.pln(l);
  }
}
```

S.o.pln(l); → [15 | 0 | 20 | 10 | 5]

S.o.pln(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 Compulsary the elements should be Homogeneous & Comparable. Otherwise we will get ClassCasteException.

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 where as 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.pln("primitive Array before Sorting:");

    for(int ai : a)
    {
      S.o.pln(ai);        10
                          5
                          20
                          11
                          6
    }

    Arrays.Sort(a);

    S.o.pln("primitive Array After Sorting:");

    for(int ai : a)
    {
      S.o.pln(ai);        5
                          6
                          10
                          11
                          20
    }

    String[] s = {"A", "Z", "B"};

    S.o.pln("object Array Before Sorting:");

    for(String a2 : s)
    {
      S.o.pln(a2);        A
                          Z
                          B
    }

    Arrays.Sort(s);

    S.o.pln("Object Array After Sorting:");
```

```
for (String a1 : s)
{
  S.o.pln(a1);   A
                 B
                 Z
}
Arrays.sort(s, new MyComparator());
S.o.pln("Object Array After Sorting by Comparator:");
for (String a1 : s)
{
  S.o.pln(a1);   Z
                 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)

Note:-

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.pln(Arrays.binarySearch(a,6));  //1
    S.o.pln(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(binarySearch(s,"Z"));  //2
    S.o.pln(binarySearch(s,"S"));  // -3
    Arrays.sort(s, new MyComparator());
```

-1	-2	-3	-4
Z	B	A	
0	1	2	

```
    S.o.pln(binarySearch(s,"Z", new MyComparator()));  // 0
    S.o.pln(binarySearch(s,"S", new MyComparator()));  // -2
    S.o.pln(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 :-

⊙ 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.

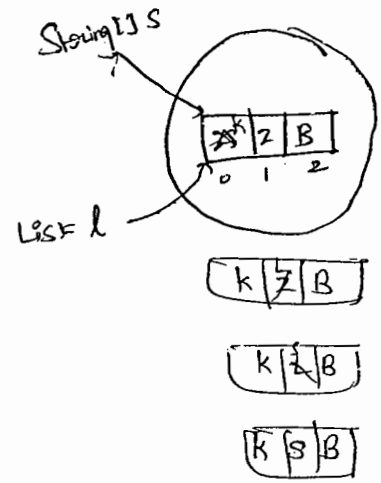
→ By using List reference we can't perform any operation which varies the size, (i.e, add & remove) otherwise we will get RuntimeException saying "unSupported-Operation-Exception" (USOE).

→ 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"

Eg:- To view Array IN LIST FORM.

ArrayAsListDemo.java

```
import java.util.*;

Class ArraysAsListDemo
{
  public static void main(String[] args)
  {
    String[] s = {"A", "Z", "B"};
    List l = Arrays.asList(s);
    S.o.pln(l);  // [A, Z, B]
    S[0] = "K";        [A, Z, B]   [K, Z, B]
    S.o.pln(l);  // [K, Z, B]
    l.Set(1, "L");     [K, Z, B]
    for(String S1 : s)
      S.o.pln(S1);  // [K, L, B]
      l.add("durga");  R.E // USOE
      l.remove(2);  R.E // USOE
      l.Set(1, "S");     [K, L, B] => [K, S, B]
      l.Set(1, 10);  R.E // ArrayStoreException
  }
}
```



175

01/05/11

Generics (1.5v)

1) Introduction
2) Generic Classes
3) Bounded types
4) Generic methods
5) Wild Card Character &
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";  ✓
      s[1] = "pavan";  ✓
      s[2] = new Student();  ✗
```

Type-Safe.

C.E :- incompatable types
found : Student
required : String.

→ Hence In the Case of Arrays we Can always give the guarantee about the ~~the~~ Type of elements. String[] array Contains only String Objects. (i.e Strings) 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 ~~go~~ 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.

Case2 :-

→ In the Case of Arrays at the time of retreival 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 retrival compulsary we should perform TypeCasting otherwise we will get CompileTimeError.

Ex:-

```
ArrayList l = new ArrayList();
l.add("durga");
:
String name1 = l.get(0);
```

C.E:- Incompatable types
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 1.5Version. Hence the main objectives of Generic Concepts are,

Hence the main objectives of Generics 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.

parameter-type.

Basetype

```
ArrayList<String> l = new ArrayList<String>();
```

→ 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 geting 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 retrival it is not required to perform any Type Casting.

```
String name1 = 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:-

parameter type.

Base 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>();

C.E:- incompatible types

found : AL< Integer>

required : List < ~~Integer~~ Object >

Conclusion 2 :-

→ for the Parameter type we can use any Class or interface name & we can't use primitive type. vilation leads to Compiletime Error.

Ex:- ArrayList <int > l = new ArrayList <int> ();

C.E:-

unexpected type

found : int

required : reference

C.E:-

unexpected type

found : int

required : reference

Generic - classes :-

→ Untill 1.4V a non-Generic Version of ArrayList Class is declared as follows.

```
class ArrayList
{
    add (Object o);
    Object get(int index)
}
```

→ The argument 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 retrival Compulsary 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 argument hence 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.

→ The return type of get() method is String, Hence at the time of retrival 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 Convension to use 'T'. e

Ex:-	Class AL<X>	Class AL<Durga>
✓	{	{
	}	}

2) 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.pln(" the Type of ob is :" +ob.getClass().getValue());
  }
  public T getOb()
  {
    return ob;
  }
```

```
Class GenDemo
{
  p.s.v.m(String[] args)
  {
    Gen<String> g1 = new Gen<String>("durga");
    g1.show();  // the type of ob is : java.lang.String
    S.o.pln(g1.getOb());  durga

    Gen<Integer> g2 = new Gen<Integer>(10);
    g1.show();  // the type of ob is : java.lang.Integer.
    S.o.pln(g2.getOb());  10
  }
}
```

Bounded Types :-

→ we Can Bound the Type parameter for a particular range By Using extends Keyword.

ex(1) :-

```
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>();
```

Exp:- Class Test<T extends Number>
{
}

→ 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 :

Exp:- ① Class Test<T implements Runnable>
✗ {
}

✗ ② Class Test<T Super Integer>
{
}

But,

→ implements Keyword purpose we Can Survive by using extends Keyword only

Exp: Class Test<T extends X>
{ → 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 it's child classes.

→ if X is an interface as the Type parameter we Can provide either X type or it's 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.

ex:-

✓ 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 Character ?

→ 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 Compiletime 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) m1(AL<? 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 it's implementation class

→ In this case also we can't add any type of elements to the list except null

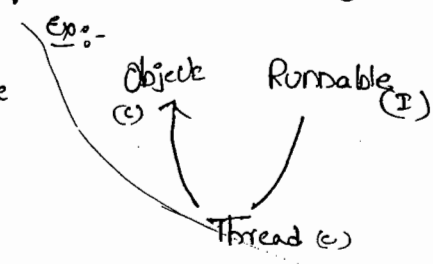
4) m1(ArrayList<? super x> l) ✓

→ 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

→ with in the method we can add only x type objects & null to the list

Ex:-



Q) which of the following declarations are valid?

✓ ① AL<String> l = new AL<String>();

✓ ② AL<?> l = new AL<String>();

✓ ③ AL<? extends String> l = new AL<String>();

✓ ④ AL<? super String> l = new AL<String>();

✓ ⑤ AL<? extends Object> l = new AL<String>();

✓ ⑥ AL< ? extends Number> l = new AL<Integer>();

✗ ⑦ AL<? extends Number> l = new AL<String>();

C.E:- Incompatible types

found : AL<String>

required: AL<? extends Number>

✗ ⑧ AL<?> l = new AL<? extends Number>();

✗ ⑨ AL<?> l = 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 Returntype.

```
Class Test <T>
{
  T ob;
  Public T get()
  {
    return ob;
  }
}
```

Class-level

```
Class Test
{
  Public <T> void m1(T t)
  {
    T ob;
  }
}
```

Method-Level

✓ ① <T extends Number>
✓ ② <T " Runnable>
✓ ③ <T " Number & Runnable>
✓ ④ <T extends Runnable & Comparable>
✗ ⑤ <T extends Number & Thread>
✗ ⑥ <T extends Runnable & Thread>

Communication with non-Generic Code :-

→ To provide Compatability with old version SUN people Compromized 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
{
  p. s.v.m (——)
  {
    AL<String> l = new AL<String>();
    l.add("A");
    l.add(10); C.E
    m1(l);
    S.o.pln(l);   [A, 10, 10.5, true]
    // l.add(10); C.E
  }
  public static void m1(AL l)
  {
    l.add(10);
    l.add(10.5);
    l.add(true);
  }
}
```

Generic area

non-Generic area

Conclusions :-

(+) Generics Concepts as applicable only at Compiletime to provide type Safety & to resolve type casting problems. At Runtime there is no Such type of Concept. Hence the following declarations are equal,

Ex:-

```
AL l = new AL();
AL l = new AL<String>();
AL l = new AL<Integer>();
```

all are equal

ex:- ArrayList l = new ArrayList<String>();

```
l.add("A");
l.add(10);
l.add(true);
S.o.pln(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();

185

Multithreading

09/04/2011

① Introduction

② The ways to define instantiate, and start a thread

③ Geting & Setting name of a thread

**④ thread priorities

⑤ the methods to prevent thread execution

- → yield()
- → join()
- → sleep()

* ⑥ Synchronization

⑦ Inter thread Communication

- → wait()
- → notify()
- → notifyAll()

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 multi-tasking:-

→ Executing Several tasks Simultaneously, where each task is a Seperate 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 Seperate independent part of the Same program is Called "thread based multitasking" & each independent part is Called "thread".

→ It is Best Suitable for "programatic Level".

→ wheather 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 areaas 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~~ in Java when Compared with C++.

(2) The Ways to define, Instanciate & Start a new thread :-

→ we Can define a thread in the following 2 ways.

(i) By extanding Thread class.

(ii) By implementing Runnable Interface.

Defining a thread by extanding Thread class :-

defining a Thread by Extanding Thread class:-

Ex:-

defining a thread

```
Class Mythread extends Thread
{
    public void run()
    {
        for(int i=0; i<=10; i++)
        {
            S.o.pln(" child thread");
        }
    }
}
```

Executing child thread (mythread)

Job of thread

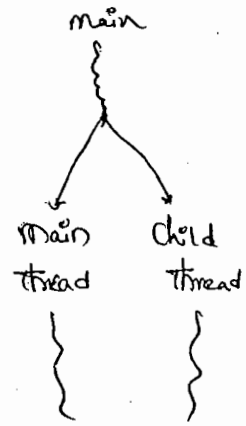
```
Class threadDemo
{
    p.s.v.m(String[] args)
    {
        MyThread t = new Mythread(); // instantiation of thread
        t.start(); // Starting of a thread
        for(int i=0; i<=10; i++)
        {
            S.o.pln(" main thread");
        }
    }
}
```

main thread

child thread

Two threads

executing main thread



Case 1:-

Thread Schedular :-

→ when ever multiple threads are waiting to get chance for execution which thread will get chance first is decided by thread schedular whose behaviour is JVM vendor dependent. Hence we can't expect exact execution order & hence exact o/p.

→ thread schedular is the part of JVM. due to this unpredictable behaviour of thread schedular we can't expect exact o/p for the above program. the following are various possible o/p.

P-1	P-2	P-3	P-4
main thread	child thread	child thread	main thread
=	=	main thread	main "
=	=	main thread	child "
=	=	=	child "
child thread	main thread	child thread	main thread
=	=	=	=
=	=	=	
=	=	main thread	
=	=	=	

Note:-

→ when ever the situation comes to multithreading the guarantee in behaviour is very less. we can tell posible 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.run() 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:-

```
child thread
child thread
8 times
main thread
10 times
```

entire o/p produced by only main thread.

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, programer 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.

Eg:-

```
class thread
{
    start()
    {
      1. Register this thread with thread scheduler & perform other
                                                initialization activities
      2. 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
{
  p.s.v.m(String[] args)
  {
    Mythread t = new Mythread();
    t.start();
  }
}
```

O/P:- no o/p printing

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 arguement 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()
    {
     S.o.pln("run()");
    }
    public void run(int i)
    {
     S.o.pln("run(int i)");
    }
}

Class ThreadDemo
{
 p.s.v.m (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()
    {
```

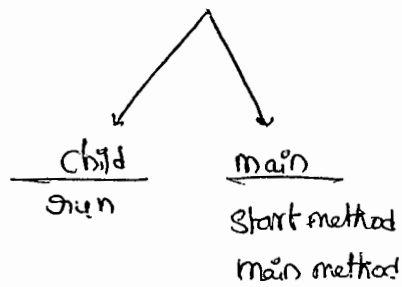
```
        S.o.pln("Start method");
        }
        public void run()
        {
          S.o.pln("run");
        }
      }
      Class ThreadDemo
      {
        p.s.v.m(String[] args)
        {
          MyThread t = new MyThread();
          t.start();
        }
      }
```

O/P:- Start method.

Case(7):-

```
      class MyThread extends Thread
      {
        public void start()
        {
          Super.start();
          S.o.pln("Start method");
        }
        public void run()
        {
          S.o.pln("run");
        }
      }
```

```
Class ThreadDemo
{
  p.s.v.m(String[] args)
  {
    Mythread t = new Mythread();
    t.Start();
    S.o.pln("main method");
  }
}
```

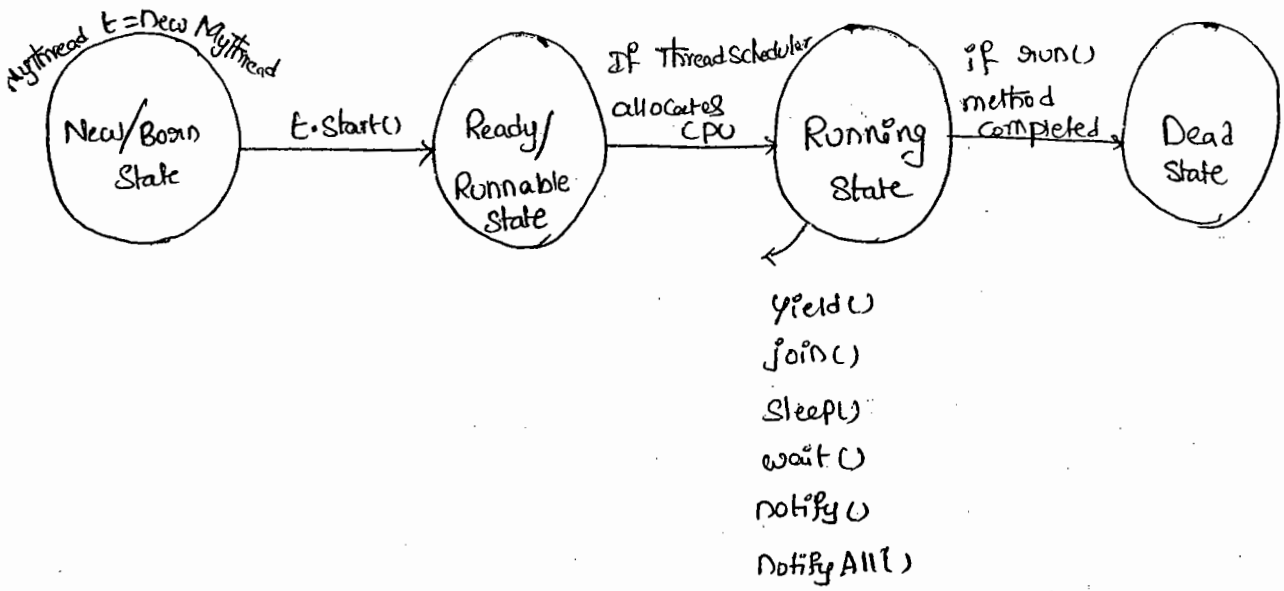


O/P:-

P-1	P-2	P-3	P-4
Start method	run	Start method	main method
run	Start method	Main 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 Dead State

*Case 9:-

→ After Starting a thread we are not allowed to restart the Same thread once again otherwise we will get IllegalRuntimeException Saying "IllegalThreadStateException".

eg:

```
Thread t = new Thread()
    t.start();
    :
    t.start(); ✗   R.E:- IllegalThreadStateException.(ITSE)
```

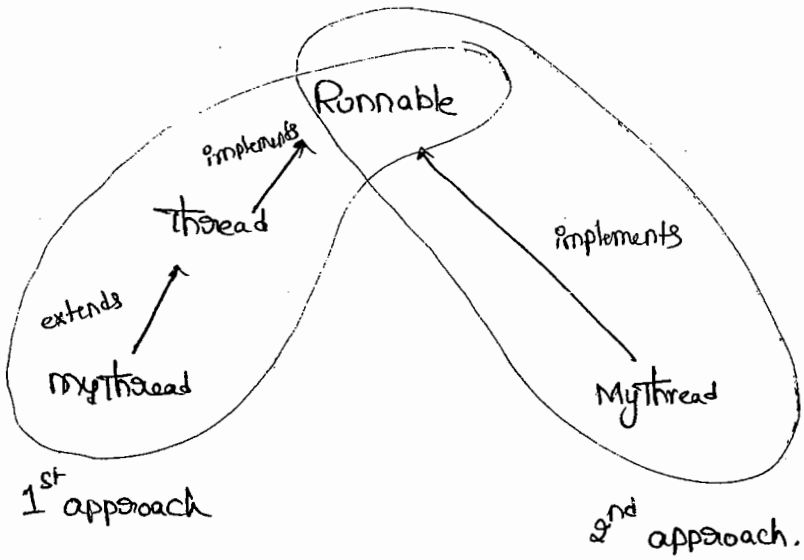
→ with in the run() if we Call Super.start() we will get the Same Run-time Exception.

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 & Contains only one method run() method.




```
Runnable
{
  run()
  {
  }
}

class Thread
{
  start()
  {
    run()
  }
}
```

Ex:-

```
Class MyRunnable implements Runnable
{
  public void run()
  {
    for (int i=0 ; i<10 ; i++)
    {
      S.o.pln("child Thread");
    }
  }
}
```

defining a thread ~~as~~ ~~Great~~

Job of thread

```
Class ThreadDemo
{
  p.s.v.m(String[] args)
  {
    MyRunnable r = new MyRunnable();
    Thread t = new Thread(r);
                 └──→ target Runnable
    t.start();
    for(int i=0; i<=10; i++)
    {
      S.o.pln("main thread");
    }
  }
}
```

→ We can't get Exact o/p & we will get mixing o/p

Case Study :-

```
MyRunnable r = new MyRunnable();
Thread t1 = new Thread();
Thread t2 = new Thread(r);
```

Case(1) :-

(i) t_1.start() :-

→ A new thread will be Created which is responsible for Execution of Thread class run().

Case(2) :- t_1.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:- r.Start():-

→ We will get Compile-time Error Saying Start() is not available in MyRunnable class

C.E:- Cannot find Symbol

Symbol : method Start()

location : class MyRunnable

Case 6: r.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() & t_2.Start()

Q) In which of the above Cases MyRunnable class run() will be executed Just like a normal method?

t_2.run() & r.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 r);

(3) Thread t = new Thread(String name);

(4) Thread t = new Thread(Runnable r, String name);

(5) Thread t = new Thread(ThreadGroup g, String name);

(6) Thread t = new Thread(ThreadGroup g, Runnable r);

(7) Thread t = new Thread(ThreadGroup g, Runnable r, String name);

(8) Thread t = new Thread(ThreadGroup g, Runnable r, String name, long stackSize);

Durga's approach to define a Thread (not recommended to use):

ex:-

```
Class MyThread extends Thread
{
  public void run()
  {
    S.o.pln("run method");
  }
}
Class Test
{
  p.s.v.m(String[] args)
  {
    MyThread t = new MyThread();
    Thread t1 = new Thread(t);
    t1.Start();
    S.o.pln("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
{
  p.s.v.m(String[] args)
  {
    S.o.pln(Thread.currentThread().getName());   // main
    Thread.currentThread().setName("prabas");
    S.o.pln(Thread.currentThread().getName());   // prabas.
  }
}
```

Note:-

→ We can get current executing thread reference by using the following method of thread class.

public static thread currentThread();

4) Thread priorities:-

→ Every thread in Java has some priority but the range of thread priorities 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~~
5) ~~Thread.HIGH_PRIORITY~~

→ Thread Schedular 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 schedular.

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); X  R.E:- IAE (IllegalArgumentException).
```

Ex:-

```
class Mythread extends thread
{
  public void run()
  {
    for(int i=0; i<10; i++)
    {
      S.o.pln("Child thread");
    }
  }
}

class ThreadPriorityDemo
{
  p.s.v.m(String[] args)
  {
    Mythread t = new Mythread();
    // t.setPriority(10);  ---> ①
    t.start();
    for(int i=0; i<=10; i++)
    {
      S.o.pln("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
≡ 10 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()

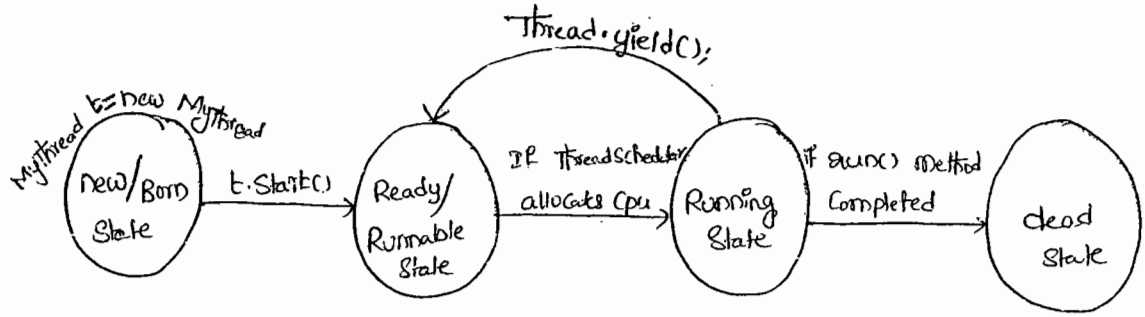
(i) yield() :-

→ 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 yieldmethod

```
public static native void yield()
```



→ The thread which is yielded, when it will get chance once again for Execution is decided by Threadschedular. & we Can't Expect Exactly.

Ex:-

```
class Mythread extends Thread
{
    public void run()
    {
        for (int i=0; i<10; i++)
        {
            Thread.yield();  ---------> ①
            S.o.pln("child thread");
        }
    }
}
class ThreadYieldDemo
{
    p.s.v.m(String[] args)
    {
        Mythread t = new Mythread();
        t.Start();
        for (int i=0; i<10; i++)
        {
            S.o.pln("main thread");
        }
    }
}
```

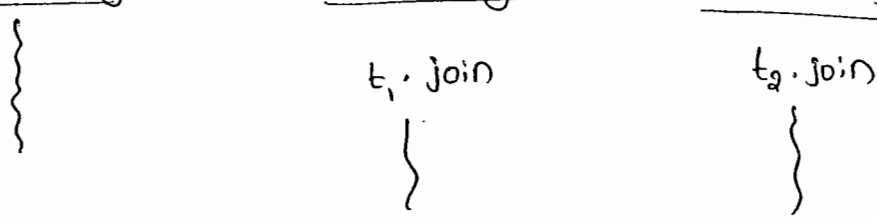
→ If we are Commenting Line① the both threads will be Executed Simultaneously & we Can't Expect Exact Execution Order.

→ If we are not Commenting Line① then the chance of Completing main thread first is high because child thread always calls yield().

ii) join() :-

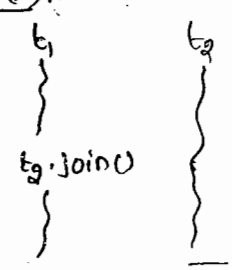
→ If a thread wants to wait until Completing Some other thread then we should go for join() method.

Ex:- (1) Venue firing (t_1) Cards printing (t_2) Cards distributing (t_3)



→ If thread t_1 Executes t_2.join() then t_1 thread will entered into waiting state until t_2 Completes. Once t_2 Completes then t_1 will Continue its execution.

ex(2):-

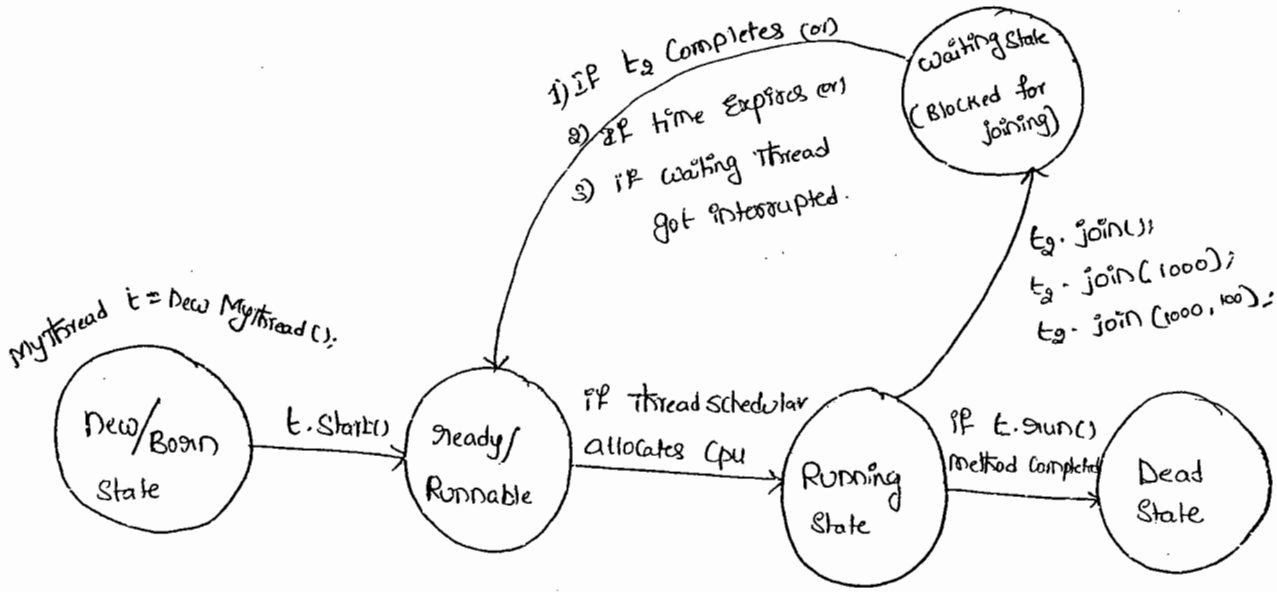


(i) public final void join() throws InterruptedException

(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 every join() throws InterruptedException. Hence, when ever we are using join() Complusary 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++)
    {
      S.o.pln(" Sitka Thread");
      try
      {
        Thread.Sleep(2000);
      }
      Catch(IE e)
      {
      }
    }
  }
Class ThreadJoinDemo
{
  p.s.v.m(String[] args) throws InterruptedException
  {
    Mythread t1 = new mythread();
    t1.Start();
    t1.join(); ———→ ①
```

```
for(int i=0 ; i<10 ; i++)
{
 S.o.pln("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
Ramathread 10 times

(iii) Sleep() :-

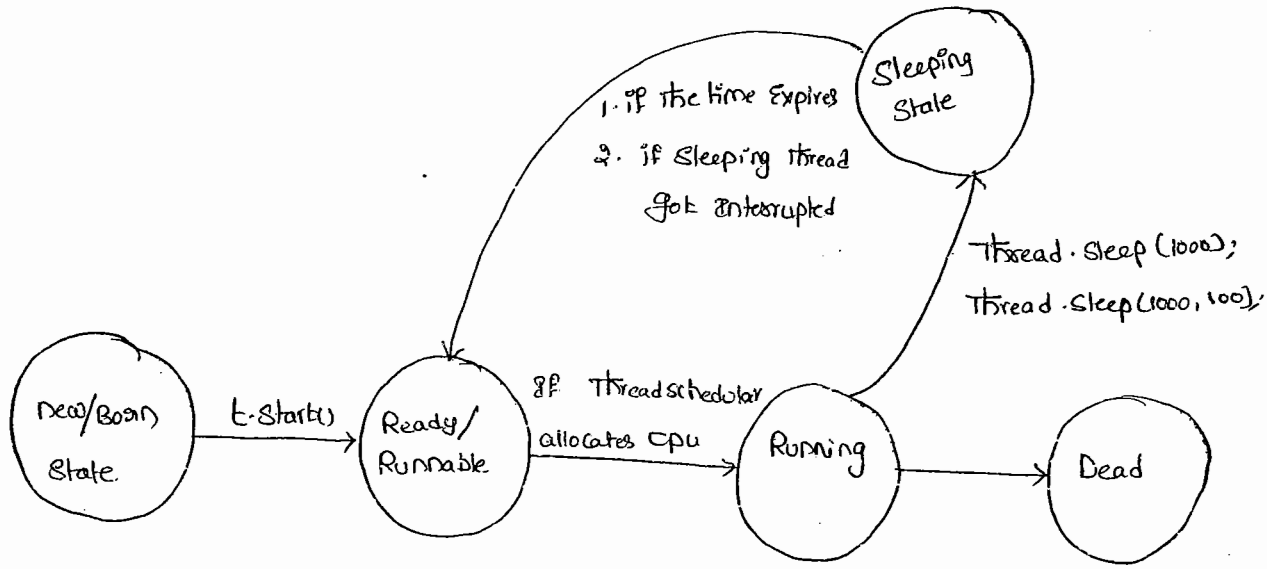
→ If a thread don't want to perform any operation for a particular amount of time (Just pausing) then we should go for Sleep()

1) Public Static void Sleep(long ms) throws InterruptedException

2) public Static void Sleep(long ms, int ns) throws InterruptedException

→ when ever we are using Sleep() method Compulsary we should handle Interrupted Exception other wise we will get Compile time Error

Static :- because sleep method Calls thread.Sleep() means Class name
t1.start(); t1 is object So it is instance (or) Non static



Ex:-

```
Class Test
{
  p.s.v.m(String[] args) throws InterruptedException
  {
    S.o.pln("Durga");
    Thread.sleep(5000);
    S.o.pln("Software");
    Thread.sleep(5000);
    S.o.pln("Solutions");
  }
}
```

Interruption of a Thread :-

* A Thread Can Interrupt another Sleeping or waiting Thread.
* for this Thread class defines interrupte() method.

```
public void interrupte()
```

Ex:-

```
Class MyThread extends Thread
{
  Public void run()
  {
   try
    {
    for (int i=0 ; i<100; i++)
    {
     S.o.pln(" Lazy Thread");
     Thread . sleep (5000);
    }
   }
  Catch (IE e)
   {
    S.o.pln(" I got Interrupted");
   }
}
Class InterruptDemo
{
  P.S.v.m (String[] args)
  {
    MyThread t = new MyThread();
      t.Start();
  →   t . interrupt();  ——→ ①
      S.o.pln(" end of main");
  }
}
```

→ If we are Commenting line ① then main thread won't Interrupte Child thread Hence both threads will be executed untill Completion

→ If we are not Commenting line① then main thread interruptes the Child thread ~~hence child thread wont Conti~~ raises Interrupted Exception.

→ In this case the o/p is

o/p:- I am Lazy thread

I got Interrupted

End of main

**Note:-

*→ We ~~can't~~ may'nt 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 untill 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.

* Comparision table for yield(), join(), Sleep() :-

Property	yield()	join()	Sleep()
1) 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 perticular amount of time (pausing) go for sleep()
2) Static	yes	No	yes
3) Is it over-loaded	No	yes	yes
4) Is it final	No	yes	No
5) Is it throws InterruptedException	No	yes	yes
6) Is it native method	yes	No	Sleep(long ms) → native Sleep(long ms, int ns) → 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. when ever we are using Synchronization then only Lock concept will come into the picture.

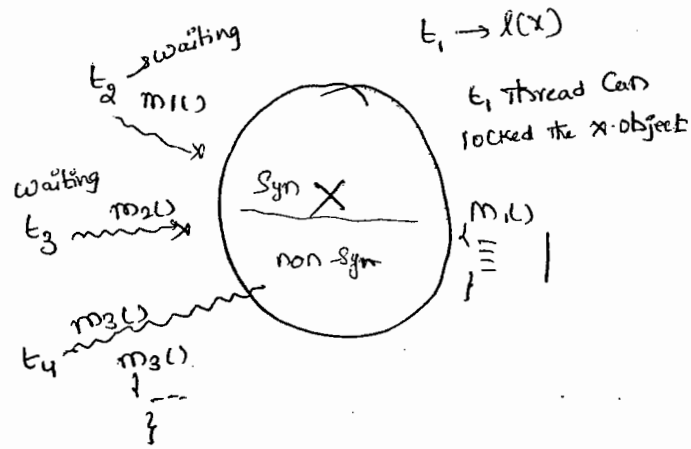
→ If a thread wants to execute any Synchronized method on the given object, first it has to get the lock of that object. Once a thread get a lock then it allowed to execute any Synchronized method on that object.

→ 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 ~~Symmult~~ 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
{
    Sync m1()
    {
    }
    Sync m2()
    {
    }
    m3()
    {
    }
}
```

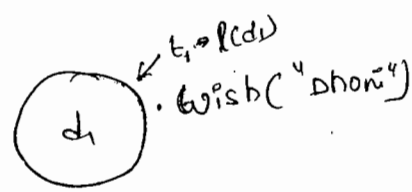


Lock:- It is an Object associated with for every Object

Ex:-

```
Class Display
{
  Public  void wish(String name)   → Synchronized
  {
    for (int i=0 ; i<10 ; i++)
    {
      S.o.print(" Good morning: ");
      try
      {
        Thread.sleep(3000);
      }
      Catche(IE e) { }
```

```
        S.O.pln(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
{
    p.s.v.m(String[] args)
    {
        Display d1 = new Display();
        MyThread t1 = new MyThread(d1, "Dhoni");
        MyThread t2 = new MyThread(d2, "Yuvaraj");
        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:-

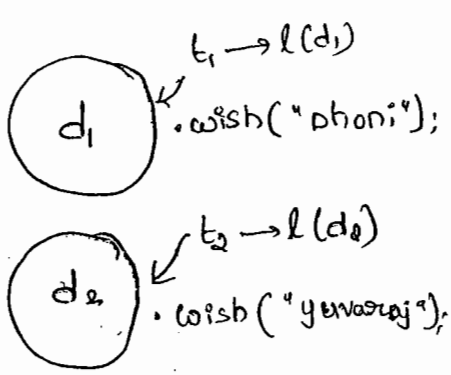
Goodmorning : Goodmorning : Dhoni
Goodmorning : yuvaraj
" : Dhoni
" "

→ 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 : yuvaraj
⋮ 10-times

Case Study:-

```
Display d1 = new Display();
Display d2 = new Display();
MyThread t1 = new MyThread(d1, "Dhoni");
MyThread t2 = new MyThread(d2, "yuvaraj");
t1.Start();
t2.Start();
```



→ Eventhough 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 Lock:-

→ 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.

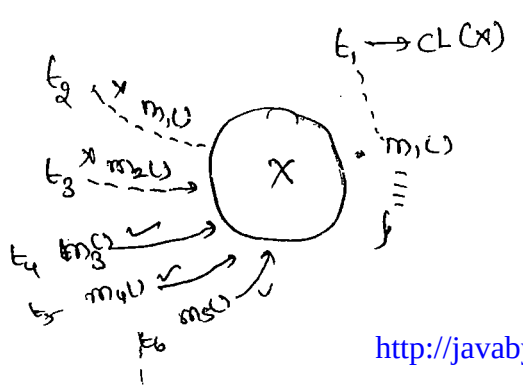
✓ 1. Normal Static methods.

✓ 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.

```
Synchronized(b)
{
  ==
  ==
  ==
}
```

→ if thread get 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

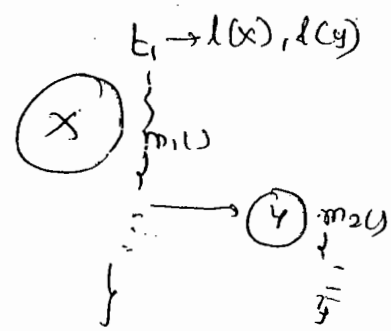
found: int

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)

Exp:

```
Class X
{
Syn m1()
{
  --
  y y = new Y();
  y.m2();
}
}

Class Y
{
Syn m2()
{
  --
}
}
```



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 ClassLevel 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)

→ 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 compulsary 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 immediatly and entered into waiting state. A thread releases the locks of only current object but not all locks. after calling notify() and notifyAll() methods thread releases the lock but may not immediatly. Except these wait(), notify(), notifyAll() there is no other case where thread releases the lock.

javap java.lang.object

method	is thread releases lock?
yield()	No
join()	No
Sleep()	No
wait()	yes
notify()	yes
notifyAll()	yes

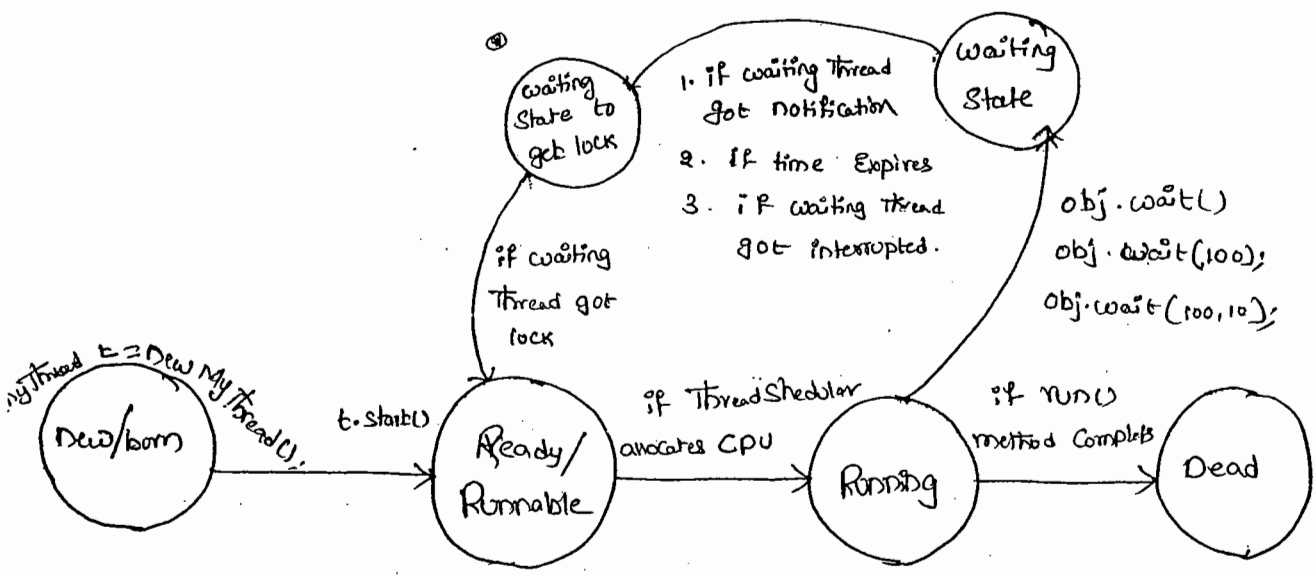
1) public final void wait() throws IE

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();
   Synchronized (b)  ---------> Thread.sleep(1000);
   {
 ① S.o.pln("main Thread trying to Call wait()");
   b.wait();   // b.wait(1000);
 ④ S.o.pln("main thread got notification").
 ⑤ S.o.pln(b.total);
   }
 }

Class ThreadB extends ThreadA
{
  int total=0;
  Public void run()
  {
   Synchronized (this)
   {
 ② S.o.pln("child Thread strats notification");
   for(int i=1 ; i<=100 ; i++)
   {
     total = total + i;
   }
 ③ S.o.pln("child thread trying to given notification");
   this.notify();
}}}
```

O/P:- main Thread Calling wait method

Child thread starts Calculation

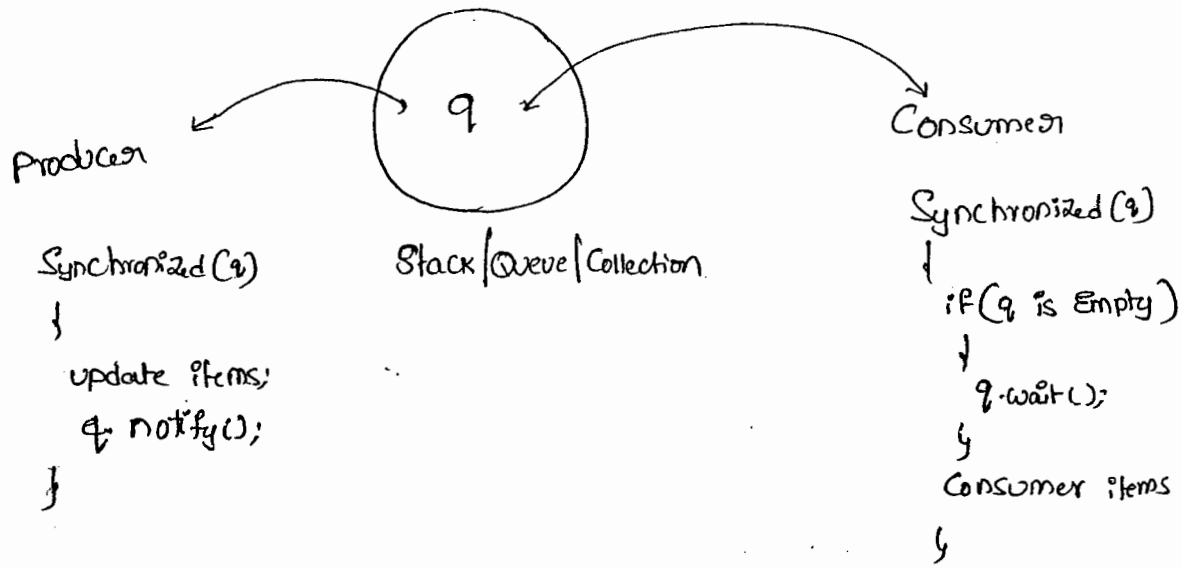
Child giving notification

Main Thread got notification

5050

Ctrl+C
KILL

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 notifying 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 s1 = new Stack();
Stack s2 = new Stack();
```

```
synchronized(s1)
{
  ==
  ==
  s2.wait();
  =
}
```

R.E: IllegalMonitorStateException

```
synchronized(s1)
{
  ==
  s1.wait();
  =
}
```

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 prevension techniques are possible.

ex:-

```
class A
{
 public synchronized void foo(B b)
 {
  S.o.pln("thread1 starts execution foo");
  try
  {
   Thread.Sleep(1000);
  }
  Catch(IE e)
  {
  }
  S.o.pln("thread1 taying to Catch b's last()");
  b.last();
 }
 Public Synchronized void last()
 {
  S.o.pln("Inside A this is last()");
 }
}
```

```
Class B
{
 Public Synchronized void bar(A a)
 {
  S.o.pln("thread2 starts bar");
  try {
      Thread.sleep(5000);
     }
  Catch(IE e) { }
 S.o.pln("thread 2 trying to call a's last");
  a.last();
 }
 Public synchronized void last()
 {
  S.o.pln("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
  }
  P.S.v.m(———).
}}{ new DeadLock()}
```

o/p :-

Thread1 Starts execution of foomethod

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 untill Completing all the Threads but this long waiting should Compulsary 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 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 saying "IllegalThreadStateException".

→ main thread is always non-Daemon & its not possible to change it's 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.pln("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.pln("end of main");
  }
}
```

→ If we are Commenting Line① 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 :→

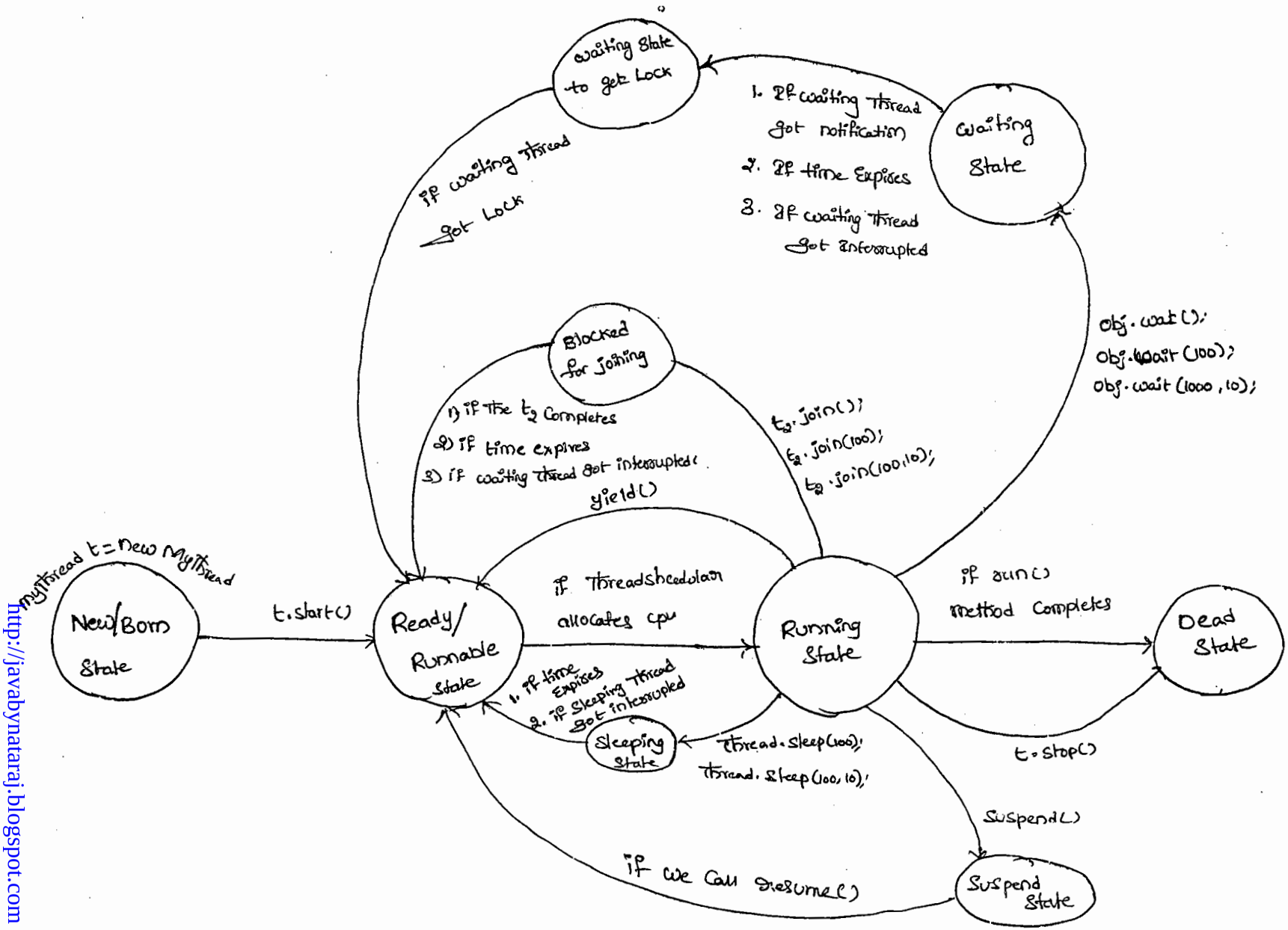
→ A Thread Can Suspend another thread by using Suspend() method.

→ A Thread Can resume a Suspended Thread by using resume() method.

→ Bot these methods are deprecated methods & Hence not recommended to use.

Q) What is a Green Thread?

Q) What is ThreadLocal?


waiting State to get Lock
1. If waiting thread got notification
2. If time Expires
3. If waiting thread got interrupted
Waiting State
if waiting thread got Lock
Blocked for joining
1) if the t2 Completes
2) if time expires
3) if waiting thread got interrupted
t2.join();
t2.join(100);
t2.join(100,10);
obj.wait();
obj.wait(100);
obj.wait(1000,10);
yield()
mythread t=new MyThread
New/Born State
t.start()
Ready/ Runnable State
if threadsheduler allocates cpu
Running State
if run() method Completes
Dead State
1. if time Expires
2. if Sleeping thread got interrupted
Sleeping State
thread.sleep(100);
thread.sleep(100,10);
t.stop()
Suspend()
if we Call resume()
Suspend State

Case 4 :-

```
Class Test
{
 public void m1(int i, float f)
 {
   S.o.pln("int-float version");
 }
 public void m1(float f, int i)
 {
   S.o.pln("float-int version");
 }
 P.S.v.m(____)
 {
   Test t1 = new Test();
   t1.m1(10, 10.5f);
   t1.m1(10.5f, 10);
 X t1.m1(10, 10);   X C.E:- reference to m1() is ambiguous
 X t1.m1(10.5f, 10.5f);  X 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.pln("Animal Version");
  }
  public void m1(monkey m)
  {
    S.o.pln("monkey version");
  }

  P.S.V.m(———)
  {
    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 Place by Compiler based on reference type and Runtime Object never play any role in overloading.

Over riding :→

03/05/11

→ "What ever 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 property()
    {
        S.o.pln("Cash + Gold + land");
    }
    Public void marry()
    {
        S.o.pln("Sobba Laxmi");
    }
}

Class C extends P
{
    Public void marry()
    {
        S.o.pln("Kajal | 3sha | 4tara | 4me");
    }
}
```

overridden method

overriding

overriding method

Ex2:-

```
Class P
{
  public void m1()
  {
    S.o.pln("parent");
  }
}
Class C extends P
{
  public void m1()
  {
    S.o.pln("child");
  }
}
Class Test
{
  p.s.v.m(——)
  {
    P p = new P();
      p.m1();  // parent
    C c = new C();
      c.m1();  // child
    P p1 = new C();
      p1.m1();  // child
  }
}
```

overriding

→ In overridding the method resolution always takes care by JVM based on runtime object & in overridding 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 develope pattern matching applications.
3. we Can develope translators like Compilers interpreters e.t.c.
4. we Can use for designing digital Circuits
5. we Can use to develope Communication protocols like TCP by IP, UDP e.t.c

ex:

```
import java.util.regex.*;
class RegExDemo
{
 p.s.v.m(String[] args)
 {
   Pattern p = Pattern.Compile("ab");
   Matcher m = p.matcher("abbbabbcbdab");
```

```
while (m.find())
{
S.o.pln (m.start() +" --- " +m.end() + " --- " +m.group());
}
}
```

O/P:- 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:-

(1) 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 " "

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@c4z#");
while (m.find())
{
S.o.p(m.start() + "----" + m.group());
}
```

x = [ab]	x = [a-z]	x = [0-9]	x = [0-9a-z]	
0 ---- a	0 --- a	1 --- 3	0 --- a	5 ---- 4
2 --- b	2 --- b	5 --- 4	1 --- 3	6 --- z
	4 --- c		2 --- b	
	6 --- z		4 --- c	

Predefined-character class :-

Space character ⟶ \s

[0-9] ⟶ \d

[0-9a-zA-Z] ⟶ \w

Anycharacter ⟶ .

Ex:

```
Pattern p = Pattern.compile("x");
Matcher m = p.matcher("a3z4@ K7#");
                       0 1 2 3 4 5 6 7 8
while(m.find())
{
 S.o.p(m.start() + "----" + m.groups());
}
```

x = \\d	x = \\w	x = \\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 — k
			7 — 7
			8 — #

Quantifiers :-

→ We can use Quantifiers to specify no. of characters to match

Ex:

1) a ⟶ Exactly one a

2) a^+ ⟶ atleast one a

3) a^* ⟶ Any no. of a's

4) a? ⟶ atmost one a

ex:-
```
Pattern p = Pattern.compile("x");
Matcher m = p.matcher("abaabaaab");
while(m.find())
{
  S.o.pln(m.start()+"-----"+m.group());
}
```

x=a	x=a+	x=a*	x=a?
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 (s) :-

Pattern class contains split() method to split given string according to a regular expression.

ex:-
```
Pattern p = pattern.compile(" |S");
String[] s = p.split("Durga Software Solutions");
for(String s1 : s)
{
  S.o.pln(s1); // Durga
                  Software
                  Solutions.
}
```

Ex(2):-

```
Pattern p = pattern.compile("\\.");        "[.]"
String[] s = p.split("www.durgajobs.com");
for (String s1 : s)
{
  S.o.pln(s1);
}
```

O/P:-
```
www
durgajobs
com
```

String class split() method :-

→ String class also contains split() to split the given string against a regular expression

Ex:-

```
String s = "www.durgajobs.com";
String[] s1 = s.split("\\.");
for (String s2 : s1)
{
  S.o.pln(s2);
}
```

```
www
durgajobs
com
```

Note:-

Pattern class split() can take target string as arguement where as String class split() can take regular expression as arguement.

StringTokenizer :-

→ We can use StringTokenizer to divide the target String into Stream of Tokens according to the

→ StringTokenizer class presenting in java.util package.

Ex:-

①
```
StringTokenizer st = new StringTokenizer("Durga Software Solutions");
while (st.hasMoreTokens())
{
  S.o.pln(st.nextToken());
}
```

o/p:-
Durga
Software
Solutions

Note:- The default regular Expression is Space

②
```
StringTokenizer st = new StringTokenizer("1,00,000", ",");
while(st.hasMoreTokens())
{
  S.o.pln(st.nextToken());
}
```

o/p:-
1
00
000

o/p:-
1
00
000

Ex(1) :-

Write a Regular Expression to represent 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:- $[a-zA-Z_][a-zA-Z0-9_][a-zA-Z0-9_]^*$

x^* (or) $x \cdot x^* = x^+$

$[a-zA-Z_][a-zA-Z0-9_]^+$

```
import java.util.regex.*;
class RegExDemo2
{
  p.s.v.m(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]))
    {
      S.o.pln(" Valid Identifier");
    }
    else
    {
      S.o.pln(" Invalid Identifier");
    }
  }
}
```

② W.a. RE to represent all valid mobile numbers

Rule:- (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]\d{9}

③ 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.

RegExp:- [a-zA-Z][a-zA-Z0-9.-]* @ [a-zA-Z0-9]+ ([.][a-zA-Z]+)+

RegExp:

" @gmail[.]com

" @(gmail|yahoo|hotmail)[.]com

Ex:-

```
import java.io.*;
import java.util.regex.*;
Class MobileExtractor1
{
  P.S.V.m(String[] args) throws IOException
  {
    PrintWriter pw = new PrintWrite("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 mail-ids are mixed with some raw data?

→ In the above Example replace regular Expression with the following mail-id regular Expression.

$$[a-zA-Z][a-zA-Z0-9]^{*}@[a-zA-Z0-9]^{+}([.][a-zA-Z]^{+})^{+}$$

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().equals(s1))
    {
      count++;
      S.o.pln(s1);
    }
  }
  S.o.pln(count);
}
}
```

P) W.a.p to delete all .bak files present in D:\durgaclass

```
import java.io.*;
import java.util.regex.*;
class FileNamesDeleter
{
  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 s1 : s)
{
  Matcher m = p.matcher(s1);
  if(m.find() && m.group().equals(s1))
  {
    Count++;
    S.o.println(s1);
    File f1 = new File(f, s1);
    f1.delete();
  }
}
S.o.println(Count);
}
}
```

Enumeration (enum)

15/5/11

→ 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 powerfull.

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();
  !
  public static final month DEC = new Month();
}
```

Month.JAN

Declaration and usage of enum :-

Ex:-

```
enum Beer
{
 KF, KO, RC, FO;
}
Class Test
{
 p.s.v.m(String[] args)
 {
   Beer b1 = Beer.KF;
   S.o.pln(b1); // KF
 }
}
```

→ We can declare enum either with in the class or outside of the class but not inside a method.

→ If we are taying to declare enum with in a method we will get Compiletime Error.

Ex:-

```
enum X
{
}
Class Y
{
}
```

✓

```
Class Y
{
  enum X
  {
  }
}
```

✓

```
Class Y
{
  public void m1()
  {
    enum X
    {
    }
  }
}
```

✗

C.E:- enum types must not be local

→ if we declare enum outside the class the applicable modifiers are public, default, strictfp.

→ if we declare enum with in a class the applicable modifiers are public, default, strictfp, private, protected, static.

enum Vs Switch Statement :-

→ until 1.4v the allowed datatypes for switch arguement are byte, short, char, int.

→ But from 1.5v onwards in addition to above the corresponding wrapper classes Byte, Short, Character, Integer, + enum type also allowed

```
switch ( )
{
}
```

1.4 V	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 arguement to switch statement.

Ep:-

```
enum Beer
{
  KF, KO, RC, FO;
}
class Test
{
```

```
p.s.v.m(--)
{
 Beer b1 = Beer.RC;
 Switch (b1)
 {
  Case KF:
        S.o.pln(" It is children's brand);
        break;
  Case KO:
        S.o.pln(" It is too lite");
        break;
  Case RC:
        S.o.pln(" It is challengers brand");
        break;
  Case FO:
        S.o.pln(" Buy one get one");
        break;
  default:
        S.o.pln(" other brands not recommended to
                                        take");
        }
    }
  }
```

O/p :- It is challengers brand.

→ If we are passing enum type as arguement 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:  ✗ C.E:- 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
{
}
enum y extends x
{
}
```

C.E:-

Cannot inherit from final x

enum types not extensible

②

```
enum x extends java.lang.Enum
{
}
```

C.E:-

③

```
enum x
{
}
class y extends x
{
}
```

C.E1:- Cannot inherit from final x

C.E2:- enum types are not extensible

④

```
class x
{
}
enum y extends x
{
}
```

C.E1:-

⑤

```
interface x
{
}
enum y implements x
{
}
```

java.lang.Enum :-

→ every enum in java ~~about~~ is always direct child class of java.lang.Enum class.

→ The power of enum is inheriting from this class only to our enum classes.

(Javap java.lang.Enum)

→ It is an abstract class & direct child class of Object class.

→ This class implements Comparable & Serializable Interfaces. hence every enum in java is bydefault Serializable and Comparable.

Values() method :-

→ we can use values() method to list out all values of enum.

eg:- Beer[] b = Beer.values();

ordinal() method :-

→ with in 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)
    {
        Beer[] b = Beer.values();
        for(Beer b1 : b)
        {
            S.o.pln(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 e.t.c... 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.

exp:- ex. enum Fish
```
{
    STAR, GOLD, GUPPY, APOLLO, KILLER;   → mandatory.
    p.s.v.m(String[] args)
    {
        S.o.pln("ENUM MAIN METHOD");
    }
}
```

> Javac Fish.java

> Java Fish

O/P:- Enum Main method.

→ In addition to Constant if we want to take any extra members Compulsary List of Constants should be in the 1st Line & should ends with ';'

Ex:- 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()
{
}
}

⑤ enum Color
{
}

→ Inside enum with out having Constant we Can't to take any Extra members, but Empty enum is always Valid.

Ex:-

enum Color
{
public void m1()
{
}
}

enum Color
{
}

Enum Class Constructors :-

→ with-in 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()
  {
    S.o.pln("Constructor");
  }
}
Class Test
{
  Public static void main(String[] args)
  {
    Beer b1 = Beer.KF;
    S.o.pln(b1);
  }
}
```

o/p:-

```
Constructor
Constructor
Constructor
Constructor
KF
```

→ We Can't Create Objects of Enum explicitly & hence we Can't Call Constructors directly.

```
Beer b = new Beer();   ✗
```

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 b1 : b)
   {
    S.o.pln(b1 + "....." + b1.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 what 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 Beer.KF.equals(Beer.RC) // False

✓ ② Beer.KF.hashCode() // ✓

✓ ③ Beer.KF ~~Beer~~ == Beer.RC ⟶ false

X ④ Beer.KF > Beer.RC

✓ ⑤ Beer.KF.ordinal > Beer.RC.ordinal

Case(1):-

```
Package pack1;
public enum Fish;
{
 STAR, GUPPY, APOLLO;
}
```

```
Package pack2;
Class Test1
{
 p.s.v.m(__)
 {
  S.o.pln(STAR);
 }
}
```

```
import static pack1.Fish.STAR;
(or)
import static pack1.Fish.*;
```

```
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.*;
```

Case2 :-

```
enum Color
{
 BLUE, RED
  {
   public void info()
   {
    S.O.pln(" Dangerous Color");
   }
  }, GREEN;

 public void info()
 {
  S.O.pln(" universal Color");
 }
}
```

```
class Test
{
  p.s.v.m(——)
  {
    Color[] c = Color.values();
    for (Color c1 : c)
    {
      c1.Info();
    }
  }
}
```

O/P :- 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 Conversions to represent dates & no.'s e.t.c

→ Our application should generate Locate Specific responses like for India people the response should be interms of Rs. (Rupees) & for the U.S people the response should be interms 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);

(2) Locale l = new Locale(String language, String Country);

Javap java.util Locale.

→ Locale class defines Several Constants to represent Some Standard Locales. We can use these Locale directly without Creating on our own.

Eg:- Locale.US Locale.ENGLISH
Locale.ITALIAN Locale.UK

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();
    S.o.pln(l1.getCountry() + "----" + l1.getLanguage());
    S.o.pln(l1.getDisplayCountry() + "----" + l1.getDisplayLanguage())
    Locale l2 = new Locale("pa", "IN");
    Locale.setDefault(l2);
    String[] S3 = Locale.getISOLanguages();
    for(String S4 : S3)
    {
      S.o.pln(S4);
    }
    String[] S4 = Locale.getISO~~Langu~~Countries();
     for(String S5 : S4)
     {
       S.o.pln(S5);
     }
     Locale[] S = Locale.getAvailableLocales();
     for(Locale S1 : S)
     {
       S.o.pln(S1.getDisplayCountry() + "---" + S1.getDisplayLanguage());
     }
 }}
```

NumberFormatt Classes :-

→ Various Countries follow Various Conversions to represent number by using NumberFormatt Class we Can formatt a number according to a particular Locale.

→ NumberFormatt class present in java.text package & it is an abstract class. Hence we Can't Create NumberFormatt object directly

```
X NumberFormatt nf = new NumberFormatt();
```

Creating NumberFormatt object for the default Locale :-

→ NumberFormatt Class defines the following methods for this

1. public static NumberFormatt 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 arguement to the above methods

Sn.
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.

Ex1:-

W.a.P to represent a Java number in Italy specific form.

```
(P) import java.text.*;
    import java.util.*;
    Class NumberFormatDemo2
    {
      P.S.V.m(____)
      {
        double d=123456.789;
        NumberFormat nf = NumberFormat.getInstance(Locale.ITALY);
        S.o.pln("Italy form is:"+nf.format(d));
      }
    }
```

O/P:- Italy form is: 123.456,789

Exp:- w.a.p to represent a java number in India, U.K & 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.p1n("India Notation is ...." + nf1.format(d));
    NumberFormat nf2 = NumberFormat.getCurrencyInstance(Locale.US);
    S.o.p1n("US Notation is ...." + nf2.format(d));
    NumberFormat nf3 = NumberFormat.getCurrencyInstance(Locale.UK);
    S.o.p1n("UK notation is ...." + nf3.format(d));
  }
}
```

any Location in India

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

Ex1.

```
NF nf = NF.getInstance();

① nf.setMaximumFractionDigits(2);
    S.o.pln(nf.format(123.4567)); // 123.45
    S.o.pln(nf.format(123.4)); //123.4

② nf.setMinimumFractionDigits(2);
    S.o.pln(nf.format(123.4567)); // 123.4567
    S.o.pln(nf.format(123.4)); // 123.40

③ nf.setMaximumIntegerDigits(3);
    S.o.pln(nf.format(123456.234)); // 456.234
    S.o.pln(nf.format(12.3456)); // 12.3456

④ nf.setMinimumIntegerDigits(3);
    S.o.pln(nf.format(123456.234)); //123456.234
    S.o.pln(nf.format(12.3456)); // 012.3456
```

Dateformat class :-

→ Various Countries follow various Conversions to represent Date.

→ By using Dateformatt class we can formatt the DATE according to a particular Locale.

→ DateFormat class is an abstract class & present in java.text pack.

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 Date format 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

229
95

Note:-
Default Style is Medium & most of the cases default Locale is US

② Public Date parse(String s) throws ParseException

To Convert Locale Specific Date Form to java Date Form.

Ex(1):-

W.a.p To display System Date in all possible Styles of U.S format

```
import java.util.*;
import java.text.*;
Class DateFormatDemo1
{
  p.s.v.m(_____)
  {
    S.o.pln("full form:" + DateFormat.getDateInstance(0).
                                    format(new Date()));
```

(or)

```
    // DateFormat df = DateFormat.getDateInstance(0);
      S.o.pln(df.format(new Date()));

    S.o.pln("Long form:" + DF.getDateInstance(1).format(new Date()))
    S.o.pln("medium form:" + DF.getDateInstance(2).format(new Date()));
    S.o.pln("SHORT form:" + DF.getDateInstance(3).format(new Date()));
  }
}
```

O/P:- full form: Thursday, February. 2. 2010

Long form : February 18, 2010

medium form : Feb 18, 2010

short form : 2/18/10

Ex 2:-

(P) w.a.p to display System Date US, UK & Italy form.

```
{
  S.o.pln("US form:" + DF.getDateInstance(0, Locale.US).format(
                                                  new Date()))
  S.o.pln("UK form:" + DF.getDateInstance(0, Locale.UK).format(new Date()));
  S.o.pln("ITALY form:" + DF.getDateInstance(0, Locale.ITALY).format(new Date()))
}
```

o/p:-

US form : Tuesday, May 18, 2010

UK form : Tuesday, 18 May 2010

ITALY form: martedì 18 maggio 2010

Getting Date-format 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 l);

Ex:-

```
S.o.pln("US form:" + DateFormat.getDateTimeInstance(0, 0, Locale.US)
                                        .format(new Date()));
```

o/p:-

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

Development

javac :-

We can use this Command to Compile a Single or group of .java files.

Syn:- javac [options] A.java/ A.java B.java *.java

- -d
- -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 where as 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 permenently by using Environment variable classpath.

→ This classpath will be preserved after system restart also

② At Command prompt level by using <u>Set</u> Command.

Set classpath = % classpath % ; D:\path >

→ This classpath will be applicable only for that particular Comand 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.

<u>Ex</u>:-
```
class Test
{
  p.s.v.m (——)
  {
    S.o.pln(" Classpath Demo");
  }
}
```

D:\Durgaclasses\> javac Test.java ↵
>java Test ↵

<u>o/p</u>:- Classpath Demo

✗ D:\ java Test ↵ <u>R.E</u>:- NoClassDeffoundError

✓ D:\> java -cp D:\Durgaclasses Test ↵ ✓

<u>o/p</u>:- classpathDemo

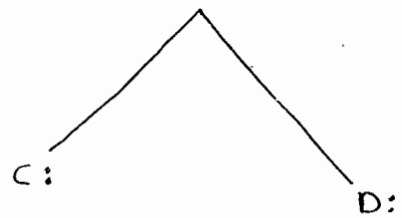
✓ G:\> java -cp D:\Durgaclasses Test ↵

<u>o/p</u>:- classpathDemo

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()
  {
    S.o.pln ("I want JOB");
  }
}
```

```
class Company
{
  p.s.v.m (___)
  {
    fresher f = new fresher();
    f.m1()
    S.o.pln("Getting JOB is very
            easy.. not required to
            worry");
  }
}
```

C:\> javac fresher.java ✓

D:\> javac Company.java ✗

C.E :- Cannot find symbol
Symbol : class fresher
location : class Company

D:\> javac -cp C: Company.java ✓

✗ D:\ java Company ←

R.E :- NoClassDefFoundError: fresher

✗ D:\ java -cp C: Company ←

R.E :- NoClassDefFoundError: Company

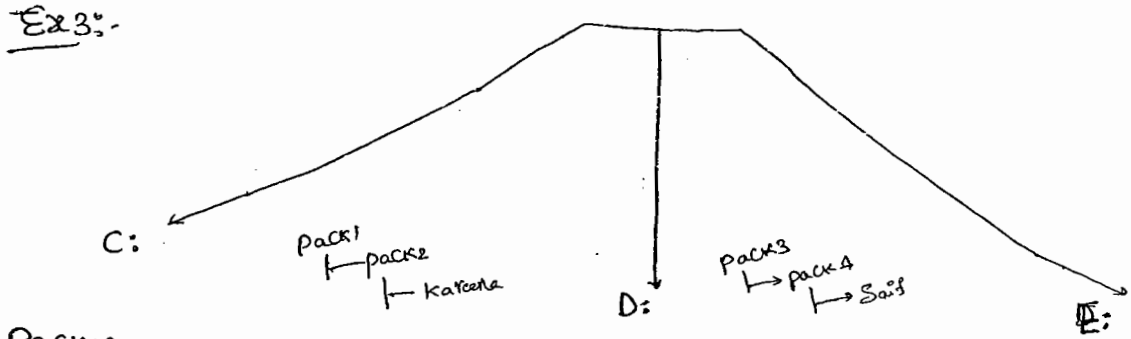
✓ D:\ java -cp D:;C: Company (or) D:\ java -cp .;C: Company

o/p :- I can JOB

Getting JOB is very easy... not required to worry.

✓ E:\ java -cp D:;C: Company

Ex3:-




```
Package pack1.pack2;
Public class Kareena
{
 public void m1()
 {
S.o.pln(" Hello Saif Can o
        please see hello
        —func");
 }
}
```

```
package pack3.pack4;
import pack1.pack2.kareena
Public class Saif
{
 public void m1()
 {
 Kareena k = new Kareena();
 k.m1();
 S.o.pln(" Not possible..As I am
        in SLIP class");
 }
}
```

```
import pack3.pack4
.Saif;
class Durga
{
 p.s.v.m(—)
 {
 Saif s = new Saif();
 s.m1();
 S.o.pln(" Can
        I help U");
 }
}
```

✓ C:\> java -d. Kareena

✓ D:\> java -d. Saif.java

C.E :- Cannot find Symbol

Symbol : class Kareena

Location : class Saif

✓ 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 : Kareena

✓ E:\> java -cp E:;D:;C: Durga

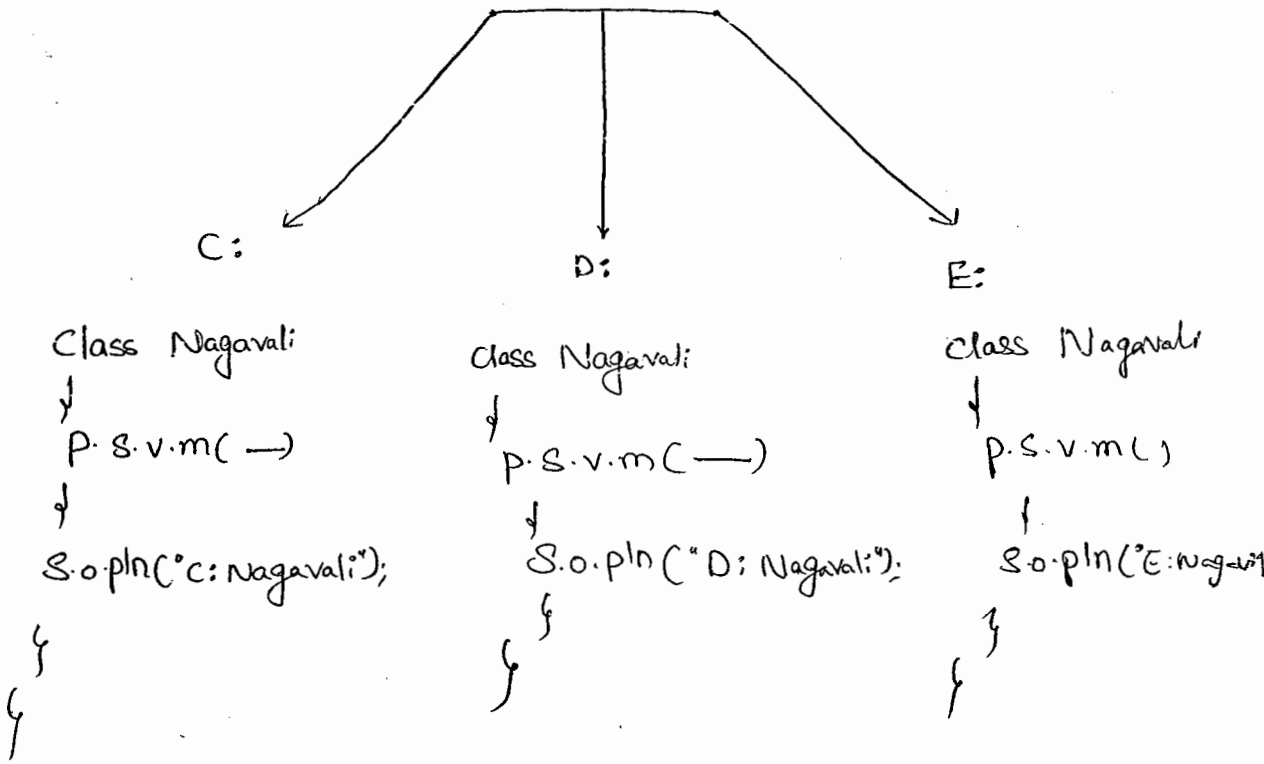
Note:-

① Compiler will check only one level dependency where as JVM will checks all levels of dependency

② If any folder structure created because of package statement it should be resolved through import statement only. & 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 to 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 :-

233 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 class path. This zip file is nothing but JAR file.

Ex(1) :-

To develope 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 archieve file)

→ It Contains a group of .class files

② war :- (web archieve file)

→ It represents a web application which may Contain Servlets, Jsps, HTMLs, CSS file, JavaScripts, e.t.c..

③ ear :- (Enterprise archieve file)

→ It represents an enterprise application which may Contains Servlets, JSPs, EJBs, JNS Components e.t.c.

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
```

```
class Bakara
{
 p.s.v.m(——)
 {
 S.o.pln(DurgaColorfulCalc.add(10,20));
 S.o.pln(DurgaColorfulCalc.multiply(10,20));
 }
}
```

✗ D:\> javac Bakara.java

✗ D:\> javac -cp c: Bakara.java

✓ D:\> javac -cp c:\durgaCalc.jar Bakara.java

✓ D:\> javac -cp .;c:\durgaCalc.jar Bakara.

O/P:- 200
400

Note:-

→ when ever we are placing a jar file in the classpath compulsary name of the jar file we should include, Just Location is not enough.

Short cut way to place jar file :-

→ If we are placing the jar file in the following Location then it is not required to set classpath explecitly bydefault it is available to Jvm & Java Compiler.

JDK
↳ JRE
↳ Lib
↳ ext
↳ *.jar

System properties :-

→ for every System persistance information will be maintain in the form of System properties. These may include O.S name, varchual 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

Space is not allowed

ex:- Java -Ddurga=SCJP Test

name of the property

Value of the property

Q) JDK vs JRE vs JVM :-

JDK:(Java developement kit) :-

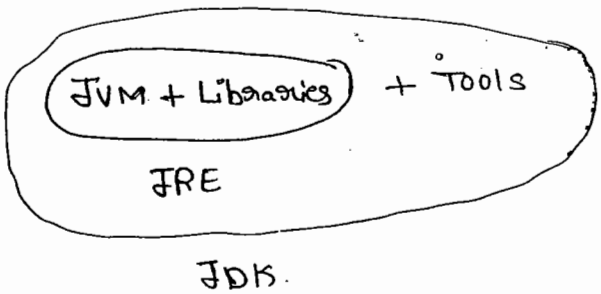
→ To develop & run Java application the required environment provided by JDK.

JRE:- (Java Runtime Environment) :-

→ To run Java application the required environment provided by JRE

JVM :-

→ This machine is responsible to execute Java program.



JDK = JRE + Tools

JRE = JVM + Libraries.

Note:-

→ On Client machine we have to install JRE, where as on the developers 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 :-

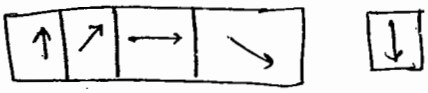
→ we can use path variable to describe the location where required Binary executables are available.

→ If we are not setting path variable then java & javac commands won't work.

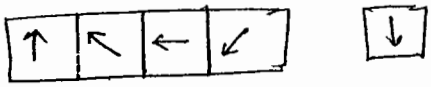
236

237

1) Clockwise

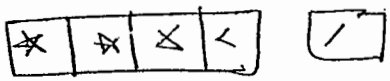


2) Anticlock

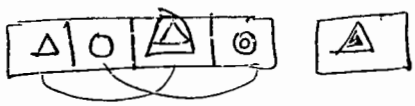


3) Increasing

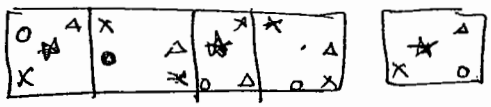
4) Decreasing



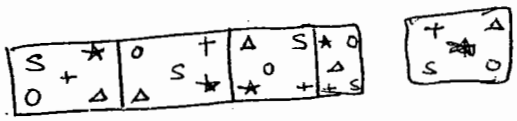
5) Alternate



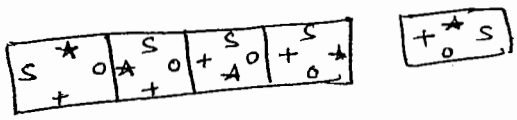
6) multiple movement



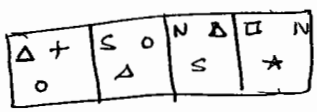
7) Rotation



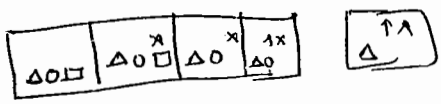
8) Interchange

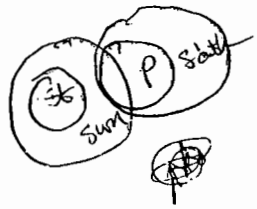


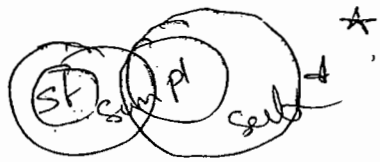
9) missing fig

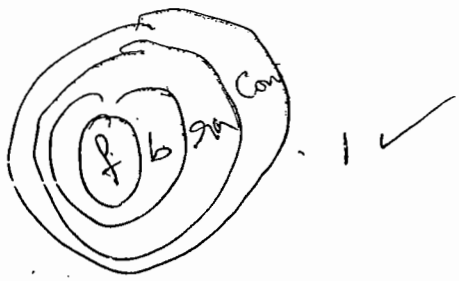


10) Substitution









238

miracle

1.5V
- → Autoboxing & unboxing ✓
- → generics.
- → var-arg ✓
- → for-each
- → enum
- → Annotations ✓
- → Queue ✓
- → Static import, not recommended ✓
- → Co-varic of return types.

Walk
↓
Jogging
↓
Running
↓
Sprinting

Siddhartha (VNR VJIT)
9951884313
Siddharthahpns@yahoo.co.in

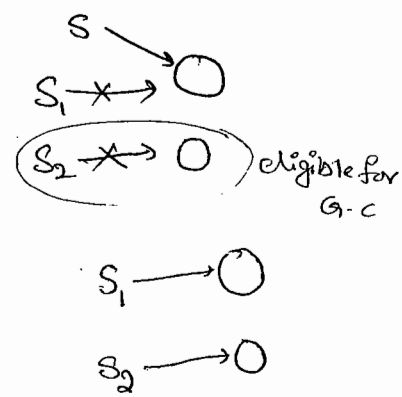
Vishnuteja.Y.S
9703340473, 9495410648
Vishnuteja87@gmail.com

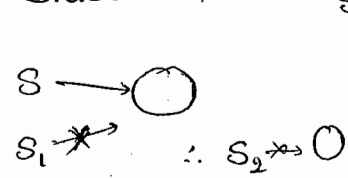
Vadu - 8179969444 (EVM)
Sluvdsarma@gmail.com

Ex 2:-

```
Class Test
{
  p.s.v.m(String[] args)
  {
    Student s = m1();
  }
  p.s.Student m1()
  {
    Student s1 = new Student();
    Student s2 = new Student();
    return s1;
  }
}
```

one object eligible for G.C →

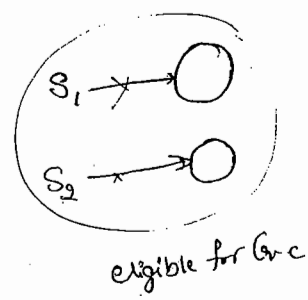




Ex 3:-

```
Class Test
{
  p.s.v.main(String[] args)
  {
    m1();
  }
  p.s.Student m1()
  {
    Student s1 = new Student();
    Student s2 = new Student();
    return s1;
  }
}
```

Two objects eligible for G.C →



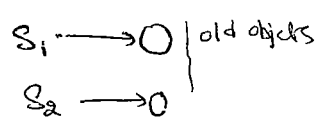
2) Reassigning the Reference variable :-

→ 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(1) :-

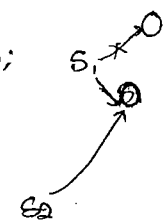
```
Student s1 = new Student();
Student s2 = new Student();
```

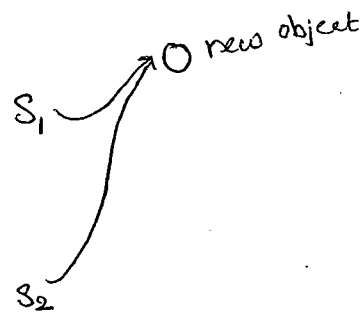
no object eligible for G.C →




```
s1 = new Student();
```

One object eligible for G.C →






```
s2 = s1;
```

Two objects eligible →

(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)
  {
    m1();
  }
  p.s.v.m1()
  {
    Student s1 = new Student();
    Student s2 = new Student();
```

2 objects eligible for G.C →

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 certain point of time for the Creation of new object sufficient memory may not be available & entire program will be collaps 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 assistent which is always runing in the background for destruction of useless objects. due to this assistant the chance of failure java program with memory problem is very rare. 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 elligible for G.C :-

→ Eventhough programmer is not responsible to destroy useless objects, it is always a good programing 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.

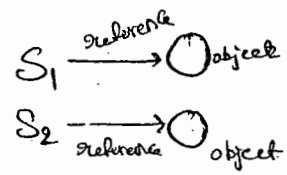
→ The following are various possible ways to make an object eligible for G.C.

(i) nullyfying the reference variable :-

→ If an object is nolonger required then assign null to all its references, then automatically that object eligible for G.C.

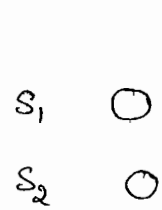
Ex:(1)

```
Student S1 = new Student();
Student S2 = new Student();
```



No objects eligible for G.C →

```
S1 = null;
```



one object eligible for G.C →

```
S2 = null;
```

two objects eligible for G.C →

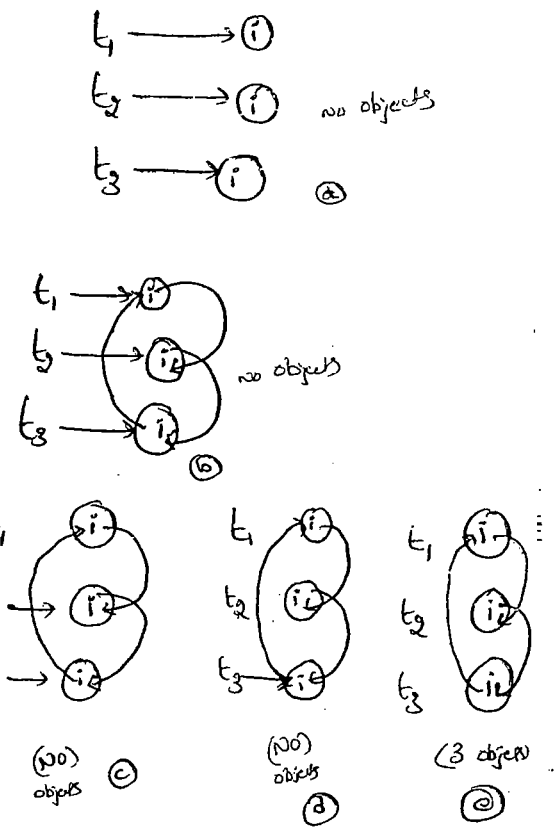
4) Island of isolation :-

Ex:-

```
Class Test
{
    Test i;
    p.s.v. main(String[] args)
    {
        Test t1 = new Test();
        Test t2 = new Test();
        Test t3 = new Test();
        t1.i = t2;
        t2.i = t3;
        t3.i = t1;
        t1 = null;
        t2 = null;
        t3 = null;
    }
}
```

No objects eligible for G.C

3 objects eligible for G.C



Note:-

→ If an object doesn't have any reference then it is always eligible for Garbage Collector.

→ Eventhough object having the reference still it is eligible for G.C sometimes (island of isolation)

The methods for requesting JVM to Run Garbage Collector:-

→ When ever we are making an object eligible for G.C it may not be destroyed by G.C immediately when ever JVM runs garbage Collector then only that object will be destroyed.

→ We Can request JVM to run garbage Collector, programatically wheather JVM accepts over request are not there is no garantee.

→ 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 freememory in the Heap,

(b) totalMemory() " total " of the Heap (Heapsize)

(c) gc() → for requesting JVM to Run garbage Collector,

Ex:-

```
class RuntimeDemo
{
  p.s.v.main(String[] args)
  {
    Runtime r = Runtime.getRuntime();
    S.o.pln(r.totalMemory());
    S.o.pln(r.freeMemory());
    for (int i=1; i<=10000; i++)
    {
      Date d = new Date();
      d=null;
    }
    S.o.pln(r.freeMemory());
    r.gc();
    System.out.println(r.freeMemory());
  }
}
```

d → ○

d ○

Q) which of the following is the properway of requested JVM to run g.c?

✓ 1) System.gc(); (System is static method

✗ 2) Runtime.gc(); (Runtime is instance method)

(g.c is applicable only static method)

✗ 3) (new Runtime()).gc();

✓ 4) Runtime.getRuntime().gc();

Note:- gc() present in the System class is a static method, where as 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 declare 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.

ex1.

```
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()
  {
    S.o.pln("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.

→ 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 (or)
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 an 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.pln("End of main");
 }
 Public void finalize()
 {
  S.o.pln("finalize method Called");
 }
}
```

o/p:

```
finalize method Called
finalize method Called
end of main
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 Servelet object Web Container always Calls destroy method, to perform Clean-up activities. But

→ It is possible to Called destroy() Explicitly from init() & Service() in this case it will be Executed Just like a normal method Call and Servelet 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.pln("end of main");
  }
```

```
public void finalize()
{
  S.o.pln("finalize method called");
  S.o.pln(10/0);
}
}
```

→ 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?

✗1) while executing finalize() all exceptions are ignored by JVM.

✓2) 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]

Eg:

```
Class finalizeDemo
{
  Static finalizeDemo s;
  p.s.v.m(String[] args) throws Exception
  {
   finalizeDemo f = new finalizeDemo();
    S.o.pln(f.hashCode());
    f = null;
    System.gc();
   Thread.sleep(5000);
   System.out.println(s.hashCode());
   S = null;
   System.gc();
   Thread.sleep(5000);
   S.o.pln("End of main method");
   }
   public void finalize()
   {
    S.o.pln("finalize method Called");
    S = this;
   }
}
```

O/P:-

```
4072869
finalize method Called
4072869
  End of main method.
```

Note:- The behaviour of the G.C is vendor dependent & hence we can't Expect exactly because of this we can't answer the following Questions Exactly.

① when JVM runs G.C exactly.

② what is the Algorithm following by g.c.?

③ In which order G.C destroys the Objects.

④ whether G.C destroys all eligible objects or not. e.t.c

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, eventhough 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 is not eligible for G.C)

→ We can resolve memory Leaks by making useless objects for G.C explicitly & by invoking G.C programmatically.

JprOBE
IBM Tivoli
HP J meter

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.ps 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 & disturbes 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 Sop is after fixing the Problem it is not required to delete assert Statements because assertions will be desabled automatically at runtime. based on our requirednent we can anable & desable assert Statements & Bydefaulk assertions are desabled.

→ Assertions Concept is applicable for developement & test environment But not for production Environment.

Assert as a Keyword & identifier :-

→ 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
{
  p.s.v.m(String[] args)
  {
    int assert=10;
    S.o.pln(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.pln(x);
 }
}
```

① Javac Test.java ✓

② Javae Test ✓

10

③ Java -ea Test ←

R.E: AssertionError

(2) Augmented Version :-

→ we can Augment some discription by using augmented version to the AssertionError.

```
assert(b) : d;
```

b → should be boolean type

d → any description, can be any type. but recommend 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.pln(x);
  }
}
```

① Javac Test.Java ✓

② Java Test ↵
10

③ Java -ea Test ↵

R.E: AssertionError: Here x value should be >10 but it is not.

Conclusion(1) :-

```
assert(e1) : e2;
```

→ e2 will be evaluated iff e1 is false. i.e if e1 is True, then e2 won't be evaluated.

ex:-

```
Class Test
{
p.s.v.m(String[] args)
{
int x=10;
assert(x==10): ++x;
S.o.p.ln(x);
}
}
```

```
assert(x>10): ++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(s):

```
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) : m1();
S.o.pln(x);
}
Public static int m1()
{
return 8888;
}
}
```

✓ javac Test.java ↵

✓ java Test ↵

10

java -ea Test

R.E: AssertionError: 8888

→ If m1() return type is void, then we will get CompileTimeError 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

NON System class	System class
✓	✓
X	✓
✓	X

→ We can use these flags in together & all these flags executed from Left to right.

Ex(2) :-

① Java -ea:pack1.A

② Java -ea:pack1.B -ea:pack1.pack2.D

③ Java -ea -da:pack1.B

Ex 2:-

```
pack1
 ├── A.class
 ├── B.class
 pack2
    ├── C.class
    └── D.class
```

→ 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 & it's Sub packages

java -ea:pack1...

→ To enable assertions in every where with in pack1 Except pack2.

java -ea:pack1... -da:pack1, pack2...

5) Appropriate & Inappropriate use of assertions :-

1) It is always Inappropriate to mix programing logic with assert Statement because there is no gamtee of execution of assert Statement at runtime.

Ex:-

```
withdraw(int x)
{
  if(x <100)
  {
   throw new IAE();
  }
}
```

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.

Ex:-
```
switch(x)
{
  case1: S.o.pln("JAN");
         break;
  Case2: S.o.pln("feb");
         break;
  :
  Case12: S.o.pln("Dec");
         break;
  default:
     assert(false);
}
```
→ R.E: A.E can be displayed.

3) It is always Inappropriate to use assertions for validating public method arguments.

4) It is always Appropriate to use assertions for validating private method arguments

5) It is always Inappropriate to use assertions for validating Command-Line arguments because these are arguments 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)
   {
    S.o.pln("I am Stupid ...b'z I am Catching AssertionError");
   }
   S.o.pln(x);
  }
 }
```

Note:-

→ It is possible to enable assertions either class wise or packagewise

250

14/02/10

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. throw.
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 disturbes normal flow of program is Called "Exception".

Ex:- Sleeping Exception, Typepunchanied Exception, filenotfound-Exception.

→ It is highly recommended to handle Exceptions. the main objective of Exception handling is "Gracefull termination of the program".

→ Exception handling doesnot 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 programing requirement is to read data from the file locating 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 Store in The Stack.

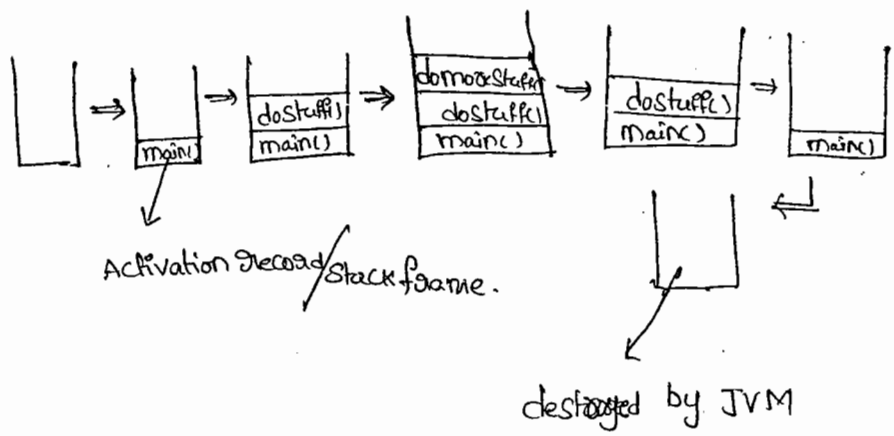
→ Each Entry in the Stack is Called "Activation Record" or "Stack frame".

→ After Completing Every method Call JVM deletes the Corresponding Entry from the Stack.

→ After Completing all method Calls, Just before Terminating the Thread JVM destroyed the Stack.

Ex:-

```
Class Test
{
p.s.v.m(String args[])
{
 doStuff();
 }
p.s.v.doStuff()
{
 doMoreStuff();
 }
 p.s.v.domoreStuff()
 {
 S.o.pln("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. discription of Exception.
3. location of Exception (Stack trace)

→ After creating Exception object, method handoveras that Exception-object to the JVM.

→ JVM checks wheather 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 Caller method & checks wheather Calleemethod Contains any handling Code or not. If the Caller method doesn't Contain any handling code, then JVM terminates that Calleemethod also abnormally & removes Corresponding Entry from the Stack.

→ This process will Continue until main() & If the main() also doesn't Contain handling Code JVM terminates the main() also abnormally & removes Corresponding Entry from Stack.

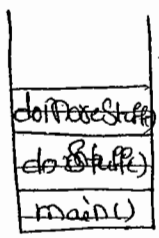
→ Just before terminating the program abnormally JVM handovers the responsibility of Exception handling to the default Exceptionhandler.

→ Default Exception handler just print Exception information to the Console in the following formatt.

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.pin(10/0);
}
}
```



Runtime Stack

Exception in thread "main" : java.lang.AE : / by zero

name of exception → java.lang.AE

description → / by zero

at Test.doMoreStuff()
at Test.doStuff()
at Test.main()

} Stack Trace.

Exception hierarchy:-

→ Throwable class acts as a root for entire Java Exception hierarcy.

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:- HallTicketMissingException,
PenNotWorkingException,
FileNotFoundException.

→ the Exceptions which are not checked by Compiler are called "un-checked Exceptions".

Ex:- BombBlastException.
ArithmaticException, FireAccidentException.

→ Wheather Exception is checked or unchecked Compulsorily it should runtime only. There is no chance of occuring at Compile time.

→ Runtime Exception and it's child classes

→ Errors & it's 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 it's 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.

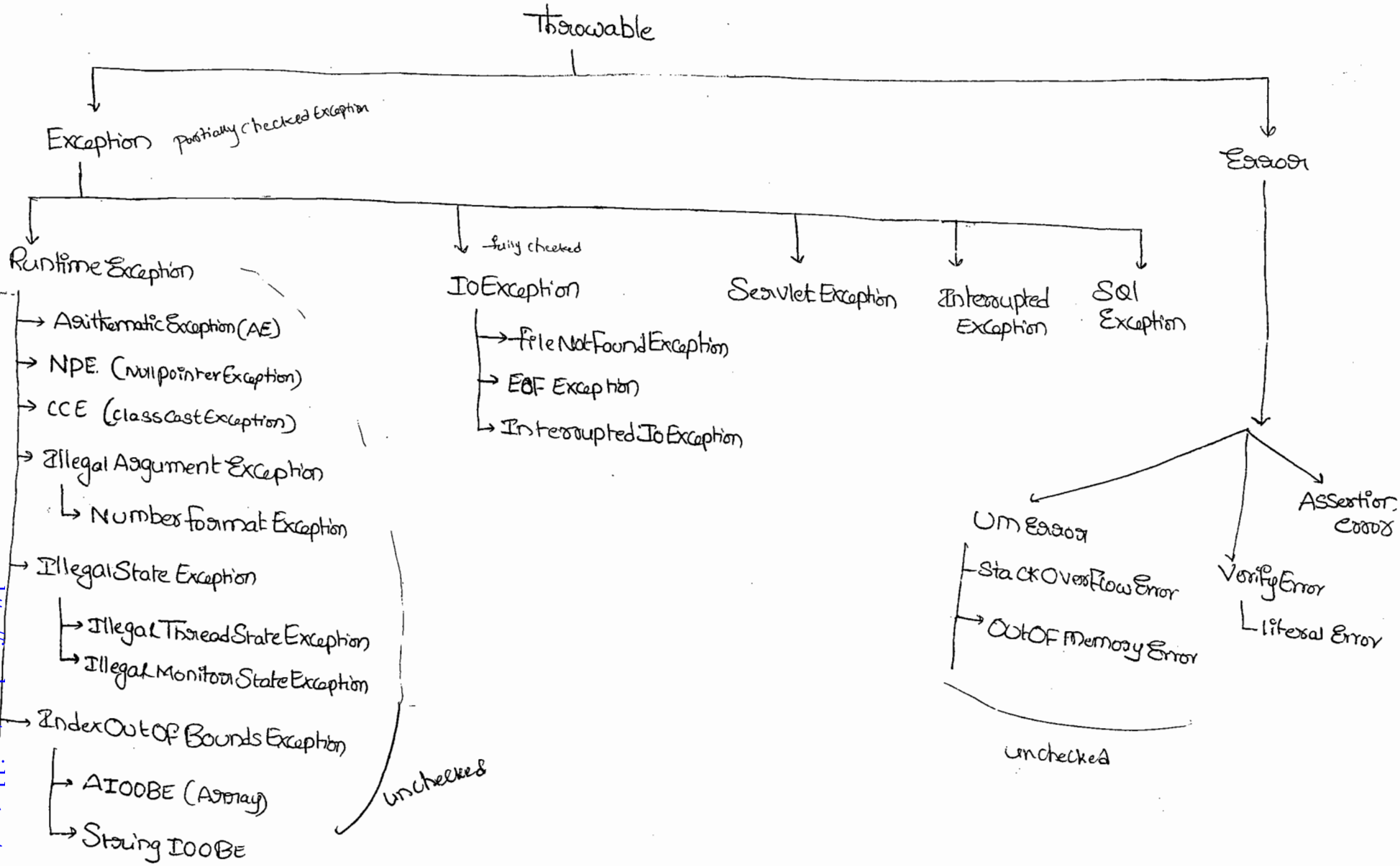
Q

(1) which of the following are checked

1) IOException : fully checked
2) Error : unchecked
3) Throwable : partially checked
4) NullPoint Exception : unchecked
5) InterruptedIOException : fully checked
6) SQLException : fully checked.

Note:

→ In Java the only Partially Checked Exceptions are 1. Exception
2. Throwable.


Throwable
Exception
partially checked Exception
Error
Runtime Exception
Arithematic Exception (AE)
NPE (NullPointerException)
CCE (ClassCastException)
Illegal Argument Exception
Number format Exception
IllegalState Exception
Illegal ThreadState Exception
Illegal MonitorState Exception
IndexOutOf Bounds Exception
AIOOBE (Array)
String IOOBE
unchecked
fully checked
IoException
FileNotFoundException
EOF Exception
InterruptedIoException
ServletException
Interrupted Exception
SQl Exception
UnError
StackOverFlowError
OutOF MemoryError
unchecked
VerifyError
literal Error
Assertion Error

Customized Exception Handling by Try-Catch:-

→ We Can maintain Risky Code with in the Try block & corresponding Handling Code inside Catch block

```
try
{
   Risky Code;
}
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("State 3");
}
}
```

o/p:- State1

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
{
 Stat 1;
 State 2;
 State 3;
}
Catch(xxx e)
{
 State 4;
}
State 5;
```

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-NT (Abnormal Termination)

Note:-

→ with in the Try block if any where 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 with in the Try block, & Length of the Try block should be as less as possible.

2. If an Exception raised at any Statement which is not part of try Then it is always Abnormal termination.

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 : discription Stack trace

(2) toString() :-

→ It prints Exception information in the following format.

Name of Exception : discription

(3) getMessage() :-

→ This method prints only discription of the Exception.

discription

Ex!-

```
Class Test
{
 p.s.v.m(String [] args)
 {
 try
  {
   S.o.pln(10/0);
  }
 Catch(A.E e)
 {
  e.printStackTrace();
  S.o.p(e); (or) S.o.pln(e.toString());
  S.o.pln(e.getMessage());
 }
}
}
```

A.E : / by zero
at test.main()

A.E : / by zero

/ 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 seperate Catch block.

Ex!-

```
try
{
 --
 --
}
Catch(Exception e)
{
 ....
}
```

(but not recommended.)

Ex(2):-

```
try
{
 --
 --
 --
}
Catch (AE e)
{
 Perform these Arithmetic operations;
}
Catch (FileNotFoundException e)
{
 use local file;
}
Catch (NPE 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 xxxxx has already been Caught"

$\boxed{\text{Child to parent is follows}}$

```
try
{
=
=
}
Catch(Exception e)
{
=
}
Catch(A.E e)
{
=
}
```

```
try
{
=
=
=
}
Catch(A.E e)
{
}
Catch(Exception e)
{
}
```

C.E :- Exception java.lang.A.E has already been Caught

finally Block :-

→ It is never recommended to define Clean-up code with in the Try blocks. because there is no garenty 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 Executeded if there is no Exception.

→ We required a place to maintain Clean-up code which should be Executed always irrespective of wheather Exception raised or not raised & wheather 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.

Ex:-

```
try
{
    risky Code;
}
catch(xxx e)
{
    handling Code;
}
finally
{
    Clean-up Code;
}
```

Ex②:-

```
Class Test
{
 p.s.v.m(String [] args)
 {
  try
  {
   S.o.pln("try");
  }
  Catch(AE e)
  {
   S.o.pln("catch");
  }
  finally
  {
   S.o.pln("finally");
  }
 }
}
```

O/P:- try
finally

```
Class Test
{
 p.s.v.m(String [] args)
 {
  try
  {
   S.o.pln("try");
   S.o.pln(10/0);
  }
  Catch(AE e)
  {
   S.o.pln("catch");
  }
  finally
  {
   S.o.pln("finally");
  }
 }
}
```

O/P:- Try
Catch
finally

```
Class Test
{
 p.s.v.m(String [] args)
 {
  try
  {
   S.o.pln("try");
   S.o.pln(10/0);
  }
  Catch(NullPointerEx e)
  {
   S.o.pln("catch");
  }
  finally
  {
   S.o.pln("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.

Ex:-
```
class Test
{
p.s.v.m(String [] args)
{
try
{
S.o.pln("try");
return;
}
Catch(A.E e)
{
S.o.pln("Catch");
}
finally
{
S.o.pln("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
{
 p.s.v.m(String [] args)
 {
 try
  {
   S.o.pln("try");
   System.exit(0);
  }
 Catch(AE e)
  {
   System.out.println("Catch");
  }
 finally
  {
   S.o.pln("finally");
  }
 }
}
```

O/P:- try

Difference b/w final, finally & finalize:-

final:-

→ It is a modifier applicable for classes, methodes & 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 (changing the value) is not allowed because, it is a constant.

finally :-

→ It is block always associated with try-catch to maintain Clean-up Code which should be Executed always irrespective of wheather exception raised or not raised & wheather handleded or not handeled.

finalize() :-

→ It is a method which should be Executed by Grabage Collector before destroying any object to perform clean-up activities.

Note :-

→ when Compare 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
{
}
Catch(xx e)
{
}
```

② ✓
```
try
{
}
Catch(xx e)   child
{
}
Catch(yy e)   parent
{
}
```

③ ✓
```
try
{
}
finally
{
}
```

④ ✗
```
try
{
}
```
C.E:- Try with out Catch or finally

⑤ ✗
```
Catch(xx e)
{
}
```
C.E:- Catch with out try

⑥ ✗
```
finally
{
}
```
C.E:- finally without try

⑦ ✗
```
try
{
}
S.o.pln("Hello");
Catch(xx e)
{
}
```
C.E:- Try without Catch or finally
C.E:- catch without try

⑧ ✓
```
try
{
}
catch(xx e)
{
}
S.o.pln("Hello");
```
A | Catch(xxe) C.E:- catch without try.

(9)
```
try
{
}
Catch(xx e)
{
}
S.o.println("Hello");
finally
{
}
```
C.E:- finally without try

(10)
```
try
{
}
Catch(xx e)
{
}
finally
{
}
finally
{
}
```
C.E:- finally without try

(11)
```
try
{
}
Catch(AE e)
{
}
Catch(Exception e)
{
}
```

(12)
```
try
{
}
Catch(exception e)
{
}
Catch(A.E 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(yy e)
  {
  }
}
```

(15)
```
try
{
}
Catch(xx e)
{
}
finally
{
  try
  {
  }
  catch(yy e)
  {
  }
}
```

(16)
```
try
{
  try
  {
  }
}
catch(xx e)
{
}
```
C.E:- try without catch or finally

(17)
```
try
{
}
finally
{
}
Catch(x e)
{
}
```
C.E: catch without try

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 Corresponsiding 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 1 or State 6, it is always abnormal terminati

Control flow in Nested try-catch-finally :-

```
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 :-

→ 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.

18/02/11

throw :-

→ Some times we can create Exception object manually & hand-over that object to the JVM explicitly by using throw keyword.

throw new ArithmeticException("/ by zero");

throw → To hand-over our created Exception object to the JVM manually.

new ArithmeticException("/ by zero") → 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.ln (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;
}
}
```

RE:- NPE

```
Cass 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 allow to write any Statement directly otherwise we will get ~~Compiletti~~ Compiletime Error Saying "unreachable Statement".

```
class Test
{
P.S.V.m(String[]args)
{
S.o.pln(10/0);
S.o.pln("Hello");
}
}
```

R.E:- AE / by zero

```
class Test
{
P.S.V.m(String[]args)
{
throw new A.E("/ by zero");
S.o.pln("Hello");
}
}
```

C.E:- unreachable Statement.

Case 3 :-

→ We can use throw keyword only for throwable type otherwise we will get compiletime Error saying Incompatiable ~~State~~ types.

```
class Test
{
 p.s.v.m(String [] args)
 {
  throw new Test();
 }
}
```

C.E: Incompatiable 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 will get compiletime Error says "unreported Exception xxxxx 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.IE 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 methodes in the case of checked Exception, to convence compiler.

→ In the case of uncheeked 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 compiles. Compulsory we should use 3 throws statements.

18/10/11

We can use throws Keyword only for Throwable types otherwise we will get Compile type Error Saying, incompatable types

```
class Test
{
  p.v.m1() throws test
  {
  }
}
```

C.E:- incompatable type
found : Test
Required : java.lang.throwable

```
Class Test extends Exception
{
  p.v.m1() throws Test
  {
  }
}
```

Case(1) :-

(Checked)

```
class Test
{
  p.s.v.m(String[]args)
  {
  throw new Exception();
  }
}
```

CE: unreported Exception java.lang. Exception must be Caught or declared to be thrown.

→ As Exception is checked Compulsory we should handle either by Try-catch or by throws Keyword

(unchecked)

```
class Test
{
  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~~ ~~not~~ 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
{
S.o.pln("Hello");
}
Catch(A.E e)
{
}
```

O/P:- Hello ✓

```
try
{
S.o.pln("Hello");
}
Catch(Exception e)
{
}
```

O/P:- Hello ✓

```
try
{
S.o.pln("Hello");
}
Catch(IOException e)
{
}
```

C.E:- Exception java.lang.IOException is never thrown in body of corresponding try statement.

```
try
{
S.o.pln("Hello");
}
Catch(InterruptedException e)
{
}
```

C.E:-

```
try ✓
{
S.o.pln("Hello");
}
Catch(Error e)
{
}
```

O/P:- 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 Compiletime Error in Exception Handling :-

① Exception xxxxx 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 Incompatable 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 "CustomizedExceptions".

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" or
                                age is already crossed marriage age").
 }
 else if(age <18)
 {
 throw new TooYoungException("Ur age is already crossed marriage
                                age ... wait some more time")
```

```
else
{
  S.o.p(" you will get match details by mail");
}
}.
```

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 devided into 2 types.

1. J.V.M Exceptions
2. programmatic Exceptions.

1. JVM Exceptions :-

→ The Exceptions which are raised automatically by the JVM when ever a particular Event occurs are Called JVM Exceptions.

Ex :- (i) ArrayIndexOutOfBoundsException.
(ii) NullPointerException.....

2. programatic Exceptions :-

→ The Exceptions which are raised Explicitly either by the programmer or by the API developer, are Called programmatic Exception.

Ex :- IllegalArgumentException, 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 without out of range index.

Ex:- int [] a = new int[10];

S.o.pln(a[0]); 0 ✓

S.o.pln(a[100]); R.E:- AIOOBE

(2) NullPointerException :-

→ It is the child class of Runtime Exception and hence it is unchecked.

→ Raised automatically by the JVM, when ever we are trying to ~~access~~ perform any operation on null.

Ex:- String s = null;

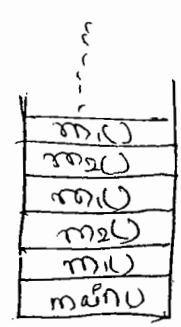
S.o.p(s.length()); RE:- NPE.

(3) StackOverFlowError :-

→ It is the child class of Error and hence it is unchecked.

→ Raised automatically by the JVM, when ever 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 [] args)
  {
    m1();
  }
}
```



R.E:- SOFE

(4) NoClassDefFoundError :-

→ It is the child class of Error and hence it is unchecked

→ raised automatically by the JVM, when ever JVM unable to find required class.

Ex: Java Sainu ↵

→ If Sainu.class file is not available then we will get R.E Saying NoClassDefFoundError.

(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("durga");
Object o = (Object) s;
```
✓

```
Object o = new Object();
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 while executing static blocks.

Ex:-

```
Class Test
{
  Static int i = 10/0;
}
```

R.E:-

ExceptionInInitializerError

Caused by java.lang.AE : / by zero.

```
Class Test
{
  Static
  {
    String s = null;
    S.o.p(ln(s.length());
  }
}
```

R.E:- ExceptionInInitializerError

Caused by java.lang.NPE

7) Illegal Argument Exception :-

→ It 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 argument

Ex:-

```
Thread t = new Thread();
t.setPriority(10);   ✓
t.setPriority(100);  ✗  R.E: IAE
```

8) NumberFormatt Exception

→ It is the child class of R.E & 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

RE ← IAE ← 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:-

One Session Expires we Can't Call any method on that object Otherwise we will get IllegalStateException.

Ex①:-

```
HttpSession Session = req.getSession();
S.o.pln(session.getId());   123u ---
```

```
Session.invalidate();
S.o.pln(Session.getId());   R.E:- ISE
```

Ex②:-

```
Thread t = new Thread();
t.start();
:
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:- Assert(false);

R.E:- AssertionError.

Exception/Error	Raised by
1. AIOOBE	JVM automatically (JVM Exception)
2. NPE	
3. SOFE	
4. NoClassDefFoundError	
5. ClassCastException	
6. ExceptionInInitializerError	
7. IllegalArgumentException	either programmer or API developer Explicitly (Programatic Exceptions)
8. NumberFormatException	
9. IllegalStateException	

Exception Propagation:-

→ The process of delegating the Resposibility Exception handling from one method to another method by using throws keyword is called Exception propagation

22/02/11

Inner Classes

→ Sometimes we can declare a class inside another class, such type of classes are called Innerclasses.

→ Innerclasses concept introduced in Java1.1 version to fix GUI bugs as the part of Eventhandling. (difficult or doubtful situation.)

→ But because of powerfull 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 Innerclass concept.

Ex(1):-

① without existing Car object, if there is no chance of existing wheel object then we should go for Inner classes.

② 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 define 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 declaration. all innerclasses are divided into 4 types

1) Normal (or) Regular Inner classes.
2) Method Local Inner classes
3) Annoymous Inner classes (without class name)
4) Static Nested Classes

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 :-

→ 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
  {
  }
}
```

Javac Outer.Java

Outer.class

Outer$Inner.class

Java Outer ↵

R.E:- NoSuchMethodError : main

Java Outer$Inner

R.E:- NoSuchMethodError : main

Ex(2):-

```
Class Outer
{
  Class Inner
  {
  }
  public static void main(String [] args)
  {
    S.o.pln("Outer class main method");
  }
}
```

O/P:- Javac Outer.java

Java Outer ↵

O/P:- OuterClass main method.

Java Outer$Inner ↵

O/P:- 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)
  {
   S.o.pln("inner class method");
  }
 }
}
```

Javac Outer.java

C.E:- Inner Classes Can't have Static declarations

23/02/11:-

Accessing Inner class Code from Static area of Outer Class:-

Ex:-
```
class Outer
{
 Class Inner
 {
  p.s.v.m1()
  {
   S.o.pln("Inner Class method");
  }
 }
 p.s.v.m(String [] args)
 {
  Outer o = new Outer();
  Outer.Inner i = O.new Inner();
```

```
        i.m1();
      }
    }
```

O/P:- Javac Outer.java ↵

Java Outer ↵

Inner class method.

```
Outer o = new Outer();
Outer.Inner i = o.new Inner();
i.m1();
```

→ Outer.Inner i = new Outer().new Inner();

→ new Outer().new Inner.m1();

Accessing Inner class Code from Instance Area of outer class:-

Eg.

```
Class Outer
  {
    class Inner
    {
      p.v.m1()
      {
        S.o.pln("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()
    {
      S.o.pln("Inner Class method");
    }
  }
}
```

```
Class Test
{
  p.s.v.m(String[] args)
  {
    Outer o = new Outer();
    Outer.Inner i = o.new
                    Inner();
    i.m1()
  }
}
```

According Inner Class Code

from static area of Outer class

(or)

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.

Ex:-

```
class Outer
{
    static int x=10;
    int y = 20;
    class Inner
    {
        public void m1()
        {
            S.o.pln(x);    10
            S.o.pln(y);    20
        }
    }
    P.S.v.m(String [] args)
    {
        new Outer().new Inner().m1();
    }
}
```

o/p:- 10
20

→ with in the Inner class <u>this</u> always pointing to current Inner class object.

→ To refer current Outer class object we have to use "<u>Outerclassname.this</u>"

" <u>Outerclassname.this</u> ".

Ex:-

Ex:-

```
class Outer
{
  int x=10;
  class Inner
  {
    int x=100;
    public void m1()
    {
      int x=1000;
      S.o.pln(x) ; 1000
      S.o.pln(this.x) ; 100
      S.o.pln(Outer.this.x) ; 10
    }
  }
  p.s.v.m(String [] args)
  {
   new Outer().new Inner().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 :-

→ Some-times 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)
      {
        S.o.pln ("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();
}
}
```

O/P:- 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 outerclass 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()
  {
    class Inner
    {
      p. void m2()
      {
        S.o.pln(x);   10
        S.o.pln(y);   20
      }
    }
    Inner i = new Inner();
    i.m2();
  }
```

if Static is there

O/P:- 20

```
p.s.v.m (String [] args)
{
 new Test().m1();
}
}
```

o/p:- 10, 20

→ from method Local Inner Class 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.

Ex:-

```
class Test
{
 int x=10;
 public void m1()
 {
  int y=20;
  class Inner
  {
   public void m2()
   {
    System.out.println(x);
    System.out.println(y);
   }
  }
  Inner i = new Inner();
  i.m2();
 }
 p.s.v.m (String [] args)
 {
  new Test().m1();
 }
}
```

→ if we declare final (final int y=20;)

o/p:- x=10
y=20

o/p:
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 Compiletime Error.

O/P:- 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 y, j.

(5) If we declare m2() as static, then which variable we can 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) Annonymus Inner Class :-

→ Sometimes we can declare a class without name also. Such type of nameless Inner classes are called Annoymus Inner classes.

→ This type of Inner classes are most Commonly used type of Inner classes.

→ There are 3 types of Annonymus Inner classes.

1. Annonymus Inner class that extends a class.
2. " " " implements an Interface.
3. " " " defined inside method arrguements.

① Annonymus Inner class that extends a class :-

Ex:-

```
Class popCorn
{
  Public void taste()
  {
    S.o.pln("Salty");
  }
  //100 more methods
}
Class Test
{
  P.S.V.m(String[] args)
  {
    PopCorn p = new PopCorn
    {
      public void taste()
      {
        S.o.pln("Sweetly");
      }
    };
    P.taste(); Sweety
    PopCorn p1 = new popCorn();
    p1.taste(); Salty
  }
}
```

Note:-

① The internal class name generated for Annoymus 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 annoymus 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.

→ ~~Pop~~

```
PopCorn p = new PopCorn()
{

};
```

→ We are creating child class for the popCorn & for that child class we are creating an object with parent reference.

Ex:

```
class Test
{
 p.s.v.m (String[] args)
 {
  Thread t = new Thread()
  {
   p.v.run()
   {
    for(int i=0 ; i<10 ; i++)
    {
     S.o.pln ("child thread");
    }
   }
  };
  t.start();
  for(int i=0 ; i<10 ; i++)
  {
   S.o.pln ("main thread");
  }
 }
}
```

→ In the above Example both main & child threads will be Executed Simultaneously & Hence we can't ~~Exe~~ exact output.

(b) Anonymous Inner class that implements an Interface :-

Ex :-

```
Class Test
{
 p.s.v.m(String [] args)
 {
  Runnable r = new Runnable()
  {
   public void run()
   {
    for (int i=0 ; i<10 ; i++)
    {
     S.o.pln("child thread");
    }
   }
  };
  Thread t = new Thread(r);
  t.start();
  for (int i=0 ; i<10 ; i++)
  {
   S.o.pln("main Thread");
  }
 }
}.
```

it is an object of Runnable

(c) Anonymous Inner Class that define Inside method argument:- 29

Eg:-

```
Class Test
{
  Public static void main(String [] args)
  {
    New Thread(new Runnable()
                {
                  public void run()
                  {
                  for (int i=0 ; i<10 ; i++)
                  {
                   S.o.pln ("child thread-i");
                  }
                }).start();

    for (int i=0 ; i<10 ; i++)
    {
     S.o.pln (" main thread -i");
    }
  }
}
```

28/02/11

General class Vs Anonymus Inner class:-

→ A General class can extend only one class at a time. where as Annonymos Innerclass also can extend only one class at a time.

→ A General class can implement any no. of Interfaces where as Annonymus Innerclass can implement only one interface at a time.

→ A General class can extend another class & can implement an interface simultaneously. where as Annonymus 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, with out 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 with out existing outer class object there may be a chance of existing Static Nested class object.

Ex:-

```
class Outer
{
    Static class Nested
    {
      public Void m1()
      {
        S.o.p1n ("static Nested class method");
    }}
```

```
public static void main(String[] args)
{
  Outer.Nested n = new Outer.Nested();
  n.m1()
}
}
```

→ With in 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)
   {
    S.o.pln("Static Nested class main method");
   }
 }
 public static void main(String[] args)
 {
  S.o.pln("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
  {
    p.v.m1()
    {
      S.o.pln(x);  ×  ——→ C.E:- Non-Static variable x can't be
      S.o.pln(y);  ✓        referenced from Static ~~class~~ Nested Content
    }
  }
}
```

Diff b/w Normal Innerclass & Static Nested Class?

Innerclass	Static Nested Class
1) Inner class object is always associated with Outerclass object. i.e without existing outerclass object there is no chance of existing Innerclass object	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 Normal Inner class we can't declare Static members	2) Inside Static Nested class we can 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 Commandprompt	3) Inside Static Nested class we can declare main() & hence we can invoke Static Nested Class directly from Commandprompt

Java.lang package

→ The most Commonly required classes & Interfaces which are required for writing any java program whether it is Simple or Complex, are encapsulated into a Seperate package which is nothing but lang package

→ It is not required to import lang package Explicitly because bydefault 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 Wrapper classes (Auto boxing & Auto unboxing)

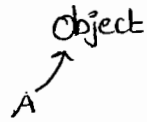
① Object :-

→ The most Common methods which are required for any java object are encapsulated into a Seperate class which is nothing but object class.

→ SUN people make this class as parent for all Java classes So that its methods are bydefault available to every Java Class Automatically

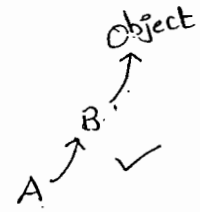
→ Every class in java is the child class of object either directly or indirectly, if our class doesnt extend any other class then only our class is direct child class of Object.

Ex:- Class A
{
}

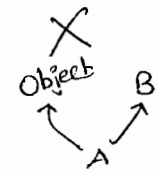


→ if our class extends any other class then our class is not direct Child class of Object. it extends object class indirectly.

Ex:- Class A extends B
{
}



multilevel Inheritance



multiple

→ Object class defines the following 11 methods

(1) public String 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 :-

→ 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 s1 = new Student("durga", 101);
    Student s2 = new Student("Sathu", 102);
    S.o.pln(s1);  (or) ⇒ S.o.pln(s1.toString());   Student@3e25a5
    S.o.pln(s2);                                   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 @ 3e2505

→ ~~To p~~

→ 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.

→ when ever 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 & in 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
{
  ~~p.s.v.m~~
  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.pln(t);      test          test@a23504
    S.o.pln(s);      durga
    S.o.pln(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 over-riding 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:- ① Case ①:-

```
Class Student
{
  ==
  ==
  public int hashCode()
  {
    return 100;
  }
  :
}
```

Case ②:-

```
Class Student
{
  ==
  ==
  public int hashCode()
  {
    return *rollno;
  }
  :
}
```

Case ①:- It is improper way of overriding hashCode() because we are generating same hashCode for every object

Case ②:- It is proper way of overriding hashCode() because we are generating a different hashCode for every object.

toString() Vs hashCode() :-

Ex:

```
Class Test
{
   int i;
   Test(int i)
   {
     this.i = i;
   }
   p.s.v.m(——)
   {
     Test t1 = new Test(10);
     Test t2 = new Test(100);
     S.o.pln(t1);      Test@1a3b2b
     S.o.pln(t2);      Test@2a4b2a
   }
}
```

object ⟶ toString()
⇓
Object ⟶ hashCode()

0-15

0
1
2
1
1
9
a(10)
b(11)
c(12)
d(13)
e(14)
f(15)

16 | 100
6 — 4
⟶
64

10

Ex (2):

```
Class Test
{
  int i;
  Test(int i)
  {
    this.i = i;
  }
  public int hashCode()
  {
    return i;
  }
  p.s.v.m(——)
  {
    Test t1 = new Test(10);
    Test t2 = new Test(100);
    S.o.pln(t1);  Test@a
    S.o.pln(t2);  Test@64
  }
}
```

object ⟶ toString()
⇓
Test ⟶ hashCode()

16 | 100
~~64~~ 4
64

16 ⇒ 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 t1 = new Test(10);
  Test t2 = new Test(100);
  S.o.pln(t1);      10
  S.o.pln(t2);      100
 }
}
```

Test → toString()

Note:-

→ if we are giving opportunity to Object Class toString() method than it will call internally hashCode() method.

→ if we are giving opportunity to over class toString() method than it may not call hashCode() method.

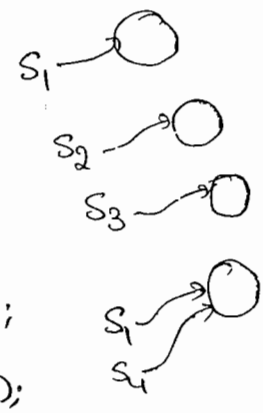
③ equals() method :-

→ 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 S1 = new Student ("durga", 101);
    Student S2 = new Student ("pavan", 102);
    Student S3 = new Student ("durga", 101);
    Student S4 = S1;
    S.o.pln (S1.equals(S2));   false
    S.o.pln (S1.equals(S3));   false
    S.o.pln (S1.equals(S4));   true
  }
}
```



→ In the above case Object class .equals() method will be executed which is always ment for reference comparision (address comparision).

→ 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 comparision instead of reference comparision we have to override .equals() method in our class.

→ When ever 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 (Hetrogeneous) equals method should return false but not ClassCastException.

(3) If we are passing Null arguement our .equals method should returns false but not a NullpointerException.

→ The following is the valid way of overriding equals() method in Student class.

Ex:

```
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;
```

```
if(name1.equals(name2) && rollno1 == rollno2)
{
  return true;
}
else
{
  return false;
}
}
Catch (CCE e)
{
  return false;
}
Catch(NPE e)
{
  return false;
}

Student s1 = new Student ("durga" 101);
Student s2 = new Student ("pavan" ,102);
Student s3 = new Student ("durga", 101);
Student s4 =s1;
S.o.pln (s1.equals( s2)); False
S.o.pln (s1.equals(s3)); True
S.o.pln (s1. equals (s4)); True
S.o.pln (s1.equals("durga")); false
S.o.pln (s1. 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 (CCE e)
    {
      return false;
    }
    Catch (CCE e)
    {
      return false;
    }
}
}
```

Relationship b/w == operator & .equals() method :-

→ if $r_1 == r_2$ is True, then r_1.equals(r_2) is always True.

→ if $r_1 == r_2$ is false, then we can't expect about r_1.equals(r_2) Exactly it may returns True or false.

→ if r_1.equals(r_2) returns True, we can't conclude anything about $r_1 == r_2$. it may returns either True or false.

→ if r_1.equals(r_2) is false, then $r_1 == r_2$ is always false.

differences b/w == operator & .equals() method :-

== operator	.equals()
① It is an operator applicable for both primitives & object references	① It is a method applicable only for object references but not for primitives.
② In the case of object references == operator is always meant for reference comparision. i.e, if two references pointing to the same object then only == operator returns T	② By default .equals() method present in object class is also meant for reference comparision only.
③ we can't override == operator for content comparision	③ we can override .equals() method for content comparision.
④ In the case of Heterogeneous type objects ~~.equal~~ == operator ~~gets~~ causes compiletime Error saying incomparable types	④ In the case of Heterogenous objects .equals() method simply return false & we won't get any compiletime or runtime Error
⑤ for any object reference r, r == null is always false.	⑤ for any object reference r, r.equals(null) is always false.

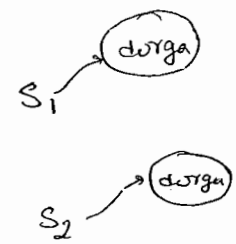
Note:-

What is the difference b/w Double equal operator (==) & .equals()

→ "==" operator is always meant for reference comparision, where as .equals() method meant for content comparision.

ex:-

```
String s1 = new String("durga");
String s2 = new String("durga");
S.o.pln(s1 == s2);   false
S.o.pln(s1.equals(s2));   true
```



→ In String, ~~All wrapper~~ classes .equals() is overriden for content comparision.

→ In StringBuffer class .equals() is not overriden for content comparision hence object class .equals() got executed which is meant for reference comparison.

→ In wrapper class .equals() is overriden for content comparision

Contract b/w .equals() & hashCode():

1. If two objects are equal by .equals() compulsary there 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.

4. 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() compulsary 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.

Q1) 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;
}
```

1) Which of the following hashCode() are said to be properly implemented.

✗ ① public int hashCode()
{
return 100;

✗ ② public int hashCode()
{
return age + (int)height;
}

✓ ③ public int hashCode()
{
return name.hashCode() + age;
}

✗ ④ public int hashCode()
{
return (int)height;
}

⑤ public int hashCode()
{
return age + name.length();
}

Note:-

To maintain a contract b/w .equals() and hashCode(), what ever the parameters we are using while over riding .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 CloneNotSupportException

```
Class Test implements cloneable
{
    int i = 10;
    int j = 20;
  P.S.V.m (——) throws CloneNotSupportException
  {
    Test t1 = new Test();
    Test t2 = (Test) t1.clone();
      t2.i = 888;
      t2.j = 999;
    S.o.pln( t1.i + " ----- " + t1.j);
  }
}
S.o.pln(t1.hashCode() == t2.hashCode()); // false
S.o.pln( t1 == t2); // false.
```

→ We can call clone() only on cloneable objects.

→ An object is said to clonable iff the corresponding class implements Clonable interface. Cloneable interface() presently java.lang package & doesn't contain any methods. It is a marker interface.

<u>Deep cloning & shallow cloning</u>:-

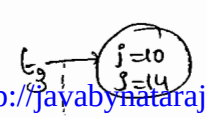
→ The process of creating just duplicate reference variable but not duplicate object is called shallow cloning.

→ The process of creating exactly duplicate independent objects is by bydefault considered as deep cloning.

ex:-
```
Test t1 = new Test();
Test t2 = t1; // shallow cloning
Test t3 = (Test)t1.clone(); // Deep cloning
```

shallow cloning

By default cloning means deep cloning.



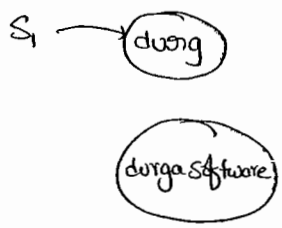
String class

24/05/11

Case(1) :-

Immutable

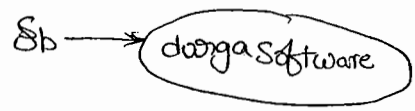
```
String s = new String("durga");
s.concate("software");
S.o.p(s);   durga
```



→ 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.o.pln(s);  // durgasoftware
```



→ ONCE we created a StringBuffer Object we Can perform any changes in the existing object. This behaviour is nothing but "mutability of String-Buffer object".

---

getClass() :-

This method returns run-time class definition of an object

eg:-

```
Test ob = new Test();
S.o.pln("Class name: " + ob.getClass().getName());
```

Case (2) :-

```
String s1 = new String("durga");
String s2 = new String("durga");
S.o.pln(s1 == s2) ; false
S.o.pln(s1.equals(s2)); true
```

→ In String class .equals() method is overridden for Content Comparision. Hence .equals() method returns True if Content is Same eventhough Objects are different.

```
StringBuffer sb1 = new StringBuffer("durga");
SB sb2 = new SB("durga");
S.o.pln(sb1 == sb2) ; false
S.o.pln(sb1.equals(sb2)); false
```

→ In StringBuffer Class .equals() method is not overridden for Content Comparision. Hence object Class .equals() method will be executed which is meant for reference comparision due to this .equals() method returns false eventhough Content is Same if objects are different

Case (3) :-

* What is the difference b/w following?

Ans

String s = new String("durga");	String s = "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	→ In this Case only one object will be Created in SCP and 's' is always pointing to that Object
heap: s → durga; SCP: durga (G.C is not allowed in SCP area)	heap: s → durga; SCP

Note:-

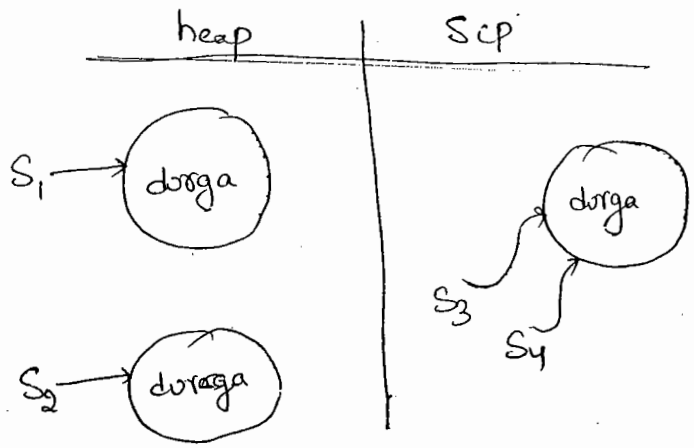
① G.C is not allowed to access in SCP area hence eventhough object doesn't have any reference variable still it is not eligible for G.C, if it is present in SCP area.

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.

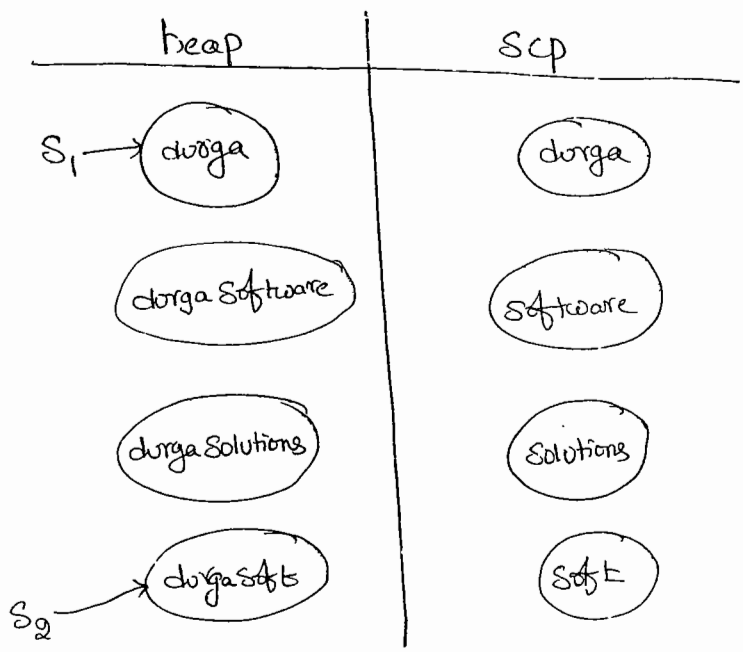
Ex ②:

```
String s1 = new String("durga");
String s2 = new String("durga");
String s3 = "durga";
String s4 = "durga";
```



Ex③ :-

```
String s1 = new String("durga");
s1.concate("software");
s1.concate("solutions");
String s2 = new s1.concate("soft");
```



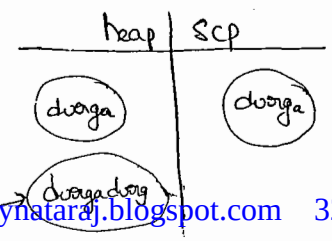
Note :-

→ for every String constant compulsary 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

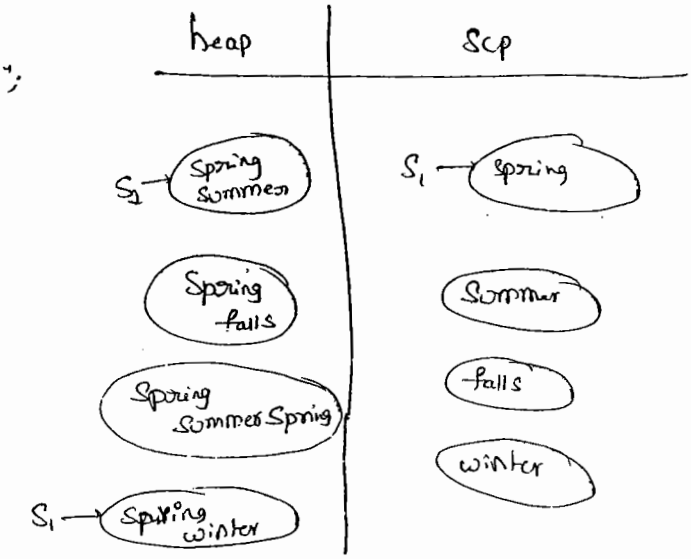
Ex④ :-

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



Ex3:-

```
String S1 = "Spring";
String S2 = S1 + "Summer";
S1.Concat("falls");
S2.Concat(S1);
S1+= "winter";
S.o.pln(S1);
S.o.pln(S2);
```



Expl:- Note:-

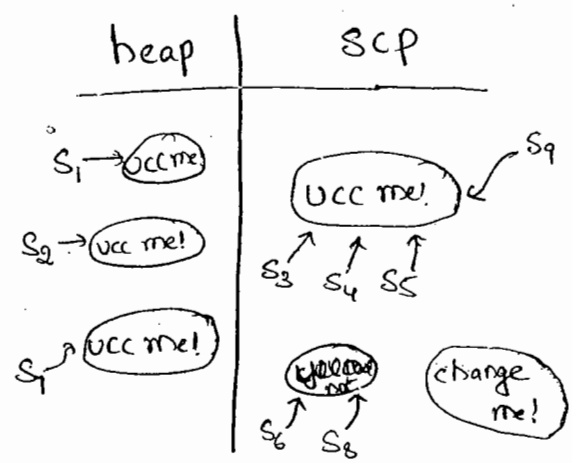
final String S = "raghu"; S is a Constant

String S = "raghu"; S is a normal variable.

Expl:-

String S1 = new String("you can't

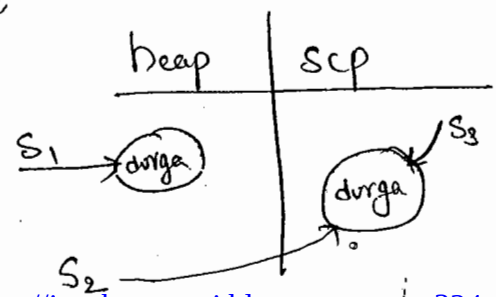
String S_1 = new String("you cannot change me!");
String S_2 = new String("you cannot change me!");
S.o.pln(S_1 == S_2); false
String S_3 = "you cannot change me!";
String S_4 = "you cannot change me!";
S.o.pln(S_1 == S_4); true
S.o.pln(S_1 == S_3); false
String S_5 = "you cannot " + "change me!";
S.o.pln(S_3 == S_5); true
String S_6 = "you cannot";
String S_7 = S_6 + "change me!";
S.o.pln(S_3 == S_7); false
final String S_8 = "you cannot";
String S_9 = S_8 + "change me!";
S.o.pln(S_3 == S_9); true
S.o.pln(S_6 == S_8); true



Interning of String :-

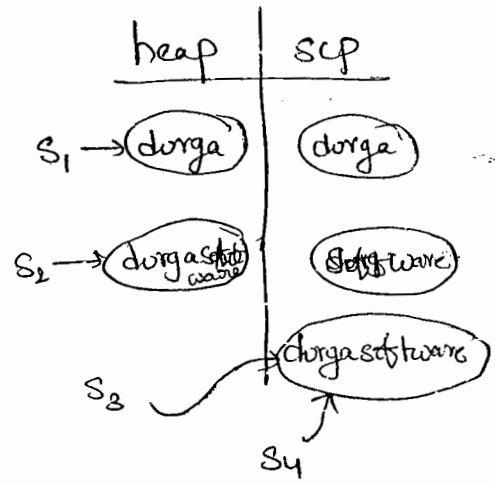
→ By using heap object reference if you want to get corresponding scp object reference then we should go for Intern().

Ex:- String S1 = new String("durga");
String S_2 = S1.intern();
S.o.pln(S_1 == S_2); false
String S_3 = "durga";
S.o.pln(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.pln(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.pln(s); abcd
```

⑤ String s = new String(byte[] b)

Eg:-
```
byte[] b = {100,101,102,103};
String s = new String(b);
S.o.pln(s); defg
```

Important methodes of String class :-

① public char charAt (int index);

Eg:- String s = "durga";

s.o.pln(s.charAt[3]); g

s.o.pln (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.pln(s); durgasoftware

→ The overloaded + , += operators also ment for Concatination Only

③ public boolean equals(Object obj) ment for Content Comparision where the Case is also important.

④ public boolean equalsIgnoreCase(String s) ment for Content Comparision where the Case is not important.

Ex:- String s = "JAVA";

s.o.pln (s.equals ("Java")); false

s.o.pln(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 method where the Case is important.

http://javabynataraj.blogspot.com

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.

Ex:- String s = "abcdefg";

S.o.pln(s.substring(3)); defg

S.o.pln(s.substring(2,6)); cdef

⑦ public int length();

eg:- String s = "aabbb";

S.o.pln(s.length); ⟶ C.E: Can't find Symbol

Symbol: variable length

location: class java.lang.String

✓ S.o.pln(s.length()); 5

Note:-

length variable applicable for arrays where as length() is applicable for string objects.

⑧ public String replace(char old, char new);

eg:- String s = "aabbb";

S.o.pln(s.replace('a','b')); bbbbb

⑨ public String toLowerCase();

⑩ public String toUpperCase();

(11) Public String trim();-

→ To remove the blank spaces present at begining & End of the String But not blankspaces present at middle of the String.

(12) public int indexOf(char ch) :-

→ It returns indexof first occurance of the Specified character

(13) public int lastIndexOf(char ch);-

Importance of String Constant pool (SCP):-

Voter Registration form

Name of Consistency : chpet

Name : Srinivas

fathername: SitaRamaiah

Age : 22

DOB :

H.NO : 9-133

Street : Ramnagar

SubStreet : Ramnagar

City : Ganapavaram

District : Guntur

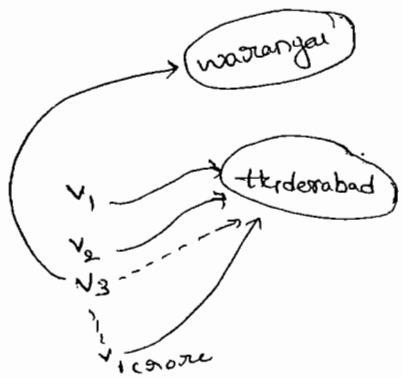
State : A.P

Country : India

PIN : 522619

Identification Name : xxxx xxxx

Submit



→ In our program if any String object required to use repetadely, it is not recommended to create a seperate 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 acheive 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 compulsary 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 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 Seperate 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?

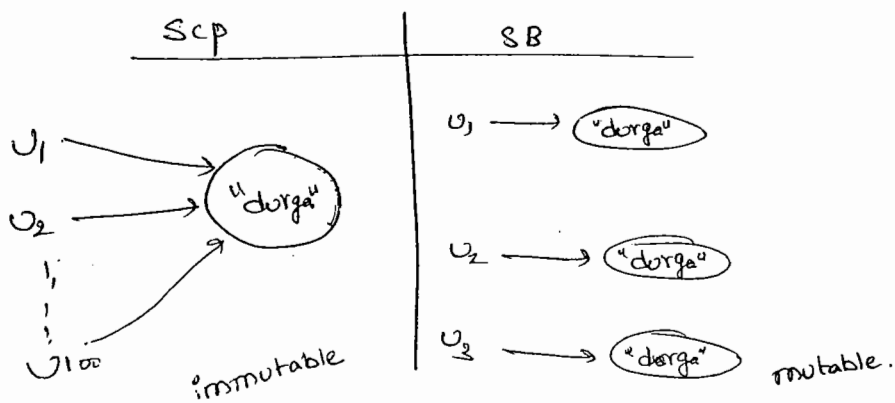
A) → Commpulsary we should make String objects as immutable.

Q) Why String objects are immutable where as 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 performaning any chang in the Existing object the remaining references coll 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 compulsory a seperate 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 & StringBuffers objects are mutable.



10/03/11

Q) Is it possible to create our own immutable class?

A) Yes,

Note:-

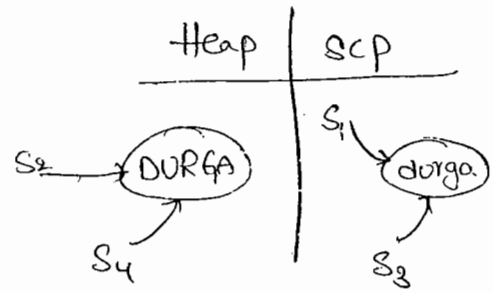
→ Once we created a String object we can't perform any changes in the existing object. If we are trying to perform any change with those changes a new object will be created on the heap.

→ Because of our runtime method call if there is a change in content then only new object will be created.

→ If there is no change in content existing object only will be reused.

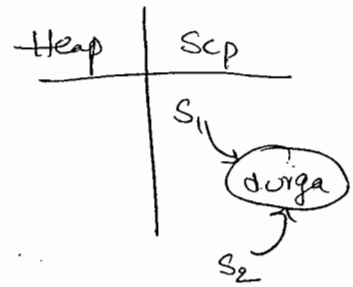
Ex:①

```
String s1 = "durga";
String s2 = s1.toUpperCase();
String s3 = s1.toLowerCase();
String s4 = s2.toUpperCase();
S.o.pln(s1 == s2); false
S.o.pln(s1 == s3); True
S.o.pln(s2 == s4) True
```



Ex②:-

```
String s1 = "durga";
String s2 = s1.toString();
S.o.pln(s1 == s2); True
```



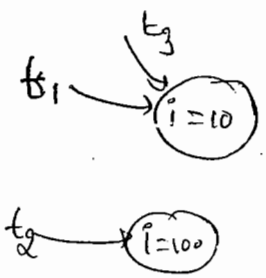
Creation of our own immutable class :-

Sol: 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.

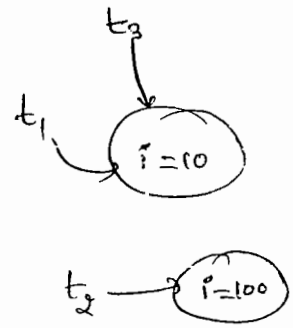
→ Because of our runtime method call if there is no change in the content then existing object only will be returned.

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 t1 = new Test (10);
Test t2 = new Test (100);
Test t3 = new Test (10);
S.o.pln (t1 == t2);   false.
S.o.pln (t1 == t3);   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 Compulsary a new object will be Created.

→ To handle this requirement Compulsary 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.pln (sb.capacity()); //16
sb.append("abcdefghijklmnop");
S.o.pln (sb.capacity()); 16
sb.append("q");
S.o.pln (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,

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("durga");

S.o.pln(sb.charAt(3)); g

S.o.pln(sb.charAt(30));
S.o.pln(sb.charAt(5));
} R.E:- StringIndexOutOfBoundsException

(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("it is exactly");
sb.append(true);
S.o.pln(sb);
```

⑥ public StringBuffer insert(int index, String s);
(int index, ~~String~~ int i);
(" boolean b);
(" double d);
⋮

ex:- StringBuffer sb = new StringBuffer("durga");

```
sb.insert(3, "srinu");
S.o.pln(sb);
```
durSrinuga.

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.pln(sb.reverse());
agrud.

⑩ public void setLength(int Length);

(10) public void setLength(int Length);

eg:- StringBuffer sb = new StringBuffer("durga123456");
sb.setLength(8);
S.o.pln(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();
S.o.pln(sb.capacity()); 5

StringBuilder :-

→ 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:**

(*)

StringBuffer	StringBuilder
① Every method is Synchronized	① No method is Synchronized.
② SB object is thread Safe. Because SB object can be accessed by only one thread at time.	② StringBuilder is not Thread Safe 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 will not 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 methodchaining

```
Sb.m1().m2().m3().m4().m5().----;
```

→ In method chaining all methods will be executed from Left to Right.

Ex :-

```
StringBuffer sb = new StringBuffer();
sb.append("durga").insert(2,"xyz").reverse().delete(2,7).append("solutions");
S.o.pln(sb); //agdsolutions
```

final vs immutable :-

300

→ If a reference variable declareded as the final then we can't reasign that reference variable to some other object.

Ex:-

```
final StringBuffer sb = new StringBuffer("durga");
sb = new StringBuffer("Software");
```

C.E:- Can't assign a value to final variable sb.

→ declaring a reference variable as final we wont get any immutability nature, in the corresponding object we can perform any type of change Eventhough reference variable declared as final.

Ex:-

```
final StringBuffer sb = new StringBuffer("durga");
sb.append("software");
S.o.pln(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 (or)

Creation of wrapper objects :-

→ All most All wrapper classes contains two constructors, one can take corresponding premitive as arguement & the other can take String as arguement.

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 properlly formatted then we will get R.E Saying NumberFormatException.

Ex:- Integer I = new Integer("ten"); R.E:- NFE

→ Float class contains 3 constructors one can take float premitive, and the other can take String & 3rd one can take double arguement

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 premitive as arrguement.

Ex:- 1) Character ch = new Character('a'); ✓

2) Character ch = new Character("a"); ✗

* Boolean class Contains two Constructors one Can take Boolean premitive as the arrguement & other Can take String as arrguement.

→ If we are passing boolean premitive as arrguement 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);

✗ Boolean B = new Boolean (True);

→ If we are passing String arrguement to the Boolean Constructor then the Case is not important & Content also not Important.

→ If the Content Case insensitive String (true), otherwise it is treted 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 ("true"); ✓ false

Wrapper classes	Corresponding Constructor arrgeement
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 b1 = new Boolean("yes");

(2) Boolean b2 = new Boolean("no");

S.o.pln(b1.equals(b2)); → true

S.o.pln(b1==b2); → false

S.o.pln(b1); false

S.o.pln(b2); false.

Note:-

→ In Every wrapper class toString() is overridden to return it's Content.

→ In Every wrapper Class .equals() is overridden for Content Comparision.

Utility Methods :-

There are 4 methods

(i) valueOf()

(ii) xxxValue()

(iii) parseXxx()

(iv) toString()

(i) valueOf() :-

→ We can use valueOf() methods for Creating wrapper object as alternative to Constructor.

Form 1:-

→ 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("p.s"); ✓

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);
```

Ex:-

```
Integer I1 = Integer.valueOf("1010", 2);
S.o.pln(I1); 10
Integer I2 = Integer.valueOf("1111", 2);
S.o.pln(I2); 15
```

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

form (3) :-

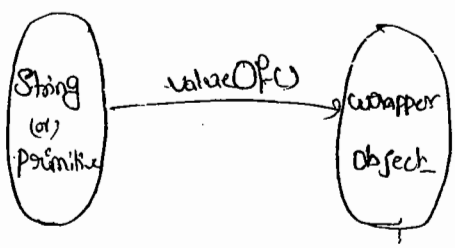
→ Every wrapper class including Character class Contains the following valueOf() to Convert primitive to Corresponding wrapper Object

```
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

302

(ii) xxxValue() :-

→ We Can use xxxValue() methods to Convert wrapper object to primitives.

→ 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();
```

eg:-

(1) Double D = new Double(130.456);

```
S.o.pln(D.byteValue()); -126
S.o.pln(D.shortValue()); 130
S.o.pln(D.intValue()); 130
S.o.pln(D.longValue()); 130
S.o.pln(D.floatValue()); 130.0
S.o.pln(D.doubleValue()); 130.0
```

charValue() :-

→ Character class Contains Char Value method to Convert Character Object to the ~~Char~~ char primitive.

* Public char charValue();

eg:- Character ch = new Character('@');

char ch1 = ch.charValue();

S.o.pln(ch1); '@'

booleanValue():

→ Boolean class contains booleanValue() to find boolean primitive for the given boolean Object.

```
public boolean booleanValue();
```

eg:- Boolean B = Boolean.valueOf("durga");

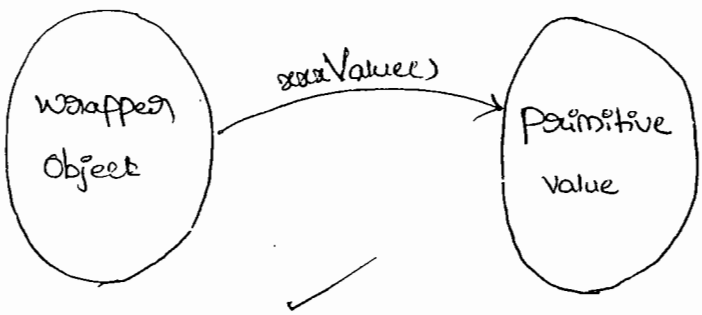
boolean b = B.booleanValue();

S.o.pln(b); false.

6x6=36
+1
+1
=38

Note:-

→ In total 38 (=6x6+1+1) xxxValue() are vailable.



(iii) parseXxx() :-

→ We Can Use parseXxx() to Convert String to Corresponding primitive.

Form1 :-

→ Every wrapper class Except Character 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");   o/p: false
```

Form2 :-

→ Every Integral type wrapper class Contains the following parseXxx() to Convert Specified radix String to Corresponding primitive.

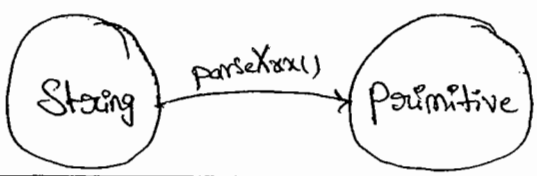
~~Eg:-~~ public static primitive parseXxx(String s, int radix);

Eg:-

```
int i = Integer.parseInt("1111", 2);
So.pln(i); 15
```

2 to 36.

Note:-



(iv) toString() :-

→ we can use toString() to convert Wrapper Object or primitive to String.

form(1) :-

→ Every wrapper class contains the following toString(), ~~to co~~ 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.o.pln(I.toString()); 10 ✓

form2 :-

→ Every wrapper 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);

form(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.p\n(s); 1111

form 4:-

→ Integer & Long classes contains the following toXxxString().

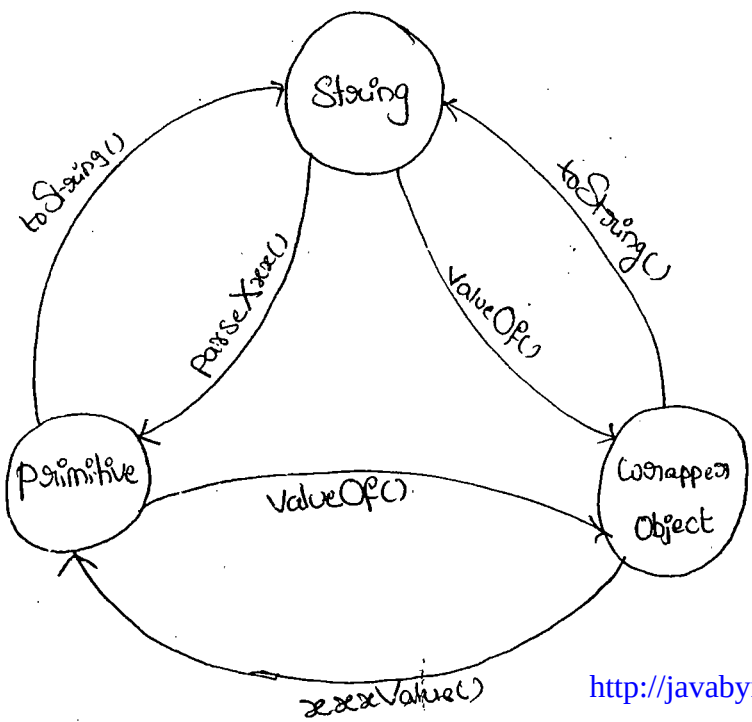
1. public static String toBinaryString(primitive p);
2. public static String toOctalString(primitive p);
3. public static String toHexString(primitive p);

Ex:- String s = Integer.toHexString(123)

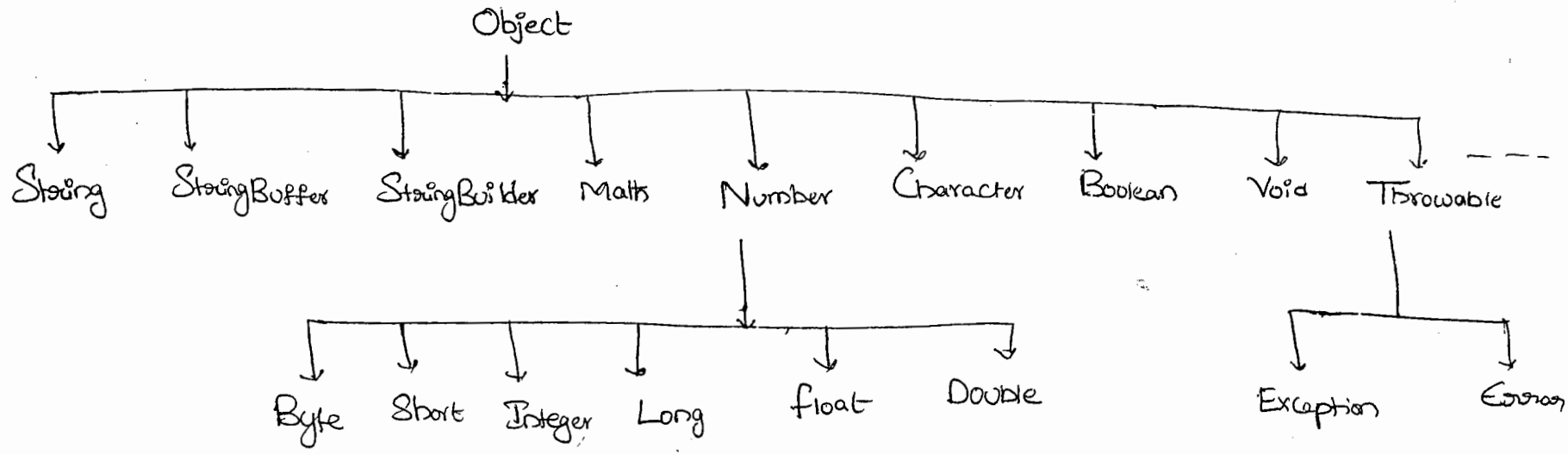
S.o.p\n(s); "7b"

16 | 123
7 – b

Dancing b/w String, Wrapper Object & primitive Value:-



Partial hierarchy of java.lang package:-



* String, StringBuffer, StringBuilder, All Wrapper Classes are final.
* The wrapper classes which are not child classes of Number are, Character & Boolean.
* The wrapper classes which are not direct child classes of Object are Byte, Short, Integer, Long, Float, Double
* Some-times we can consider Void also as Wrapper Classes
* In addition to String object all wrapper objects are Immutable.

16-3-11

305

Autoboxing & Autounboxing :- (1.5V)

→ 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); ✗ C.E!.

Integer I = new Integer(10);
l.add(I); ✓

② Boolean B = new Boolean(true);

```
if(B)
{
  S.o.pln("Hello");
}
```

C.E!- Incompatible types
found : Boolean
required : boolean

```
boolean b = B.booleanValue();
if(b)
{
  S.o.pln("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".

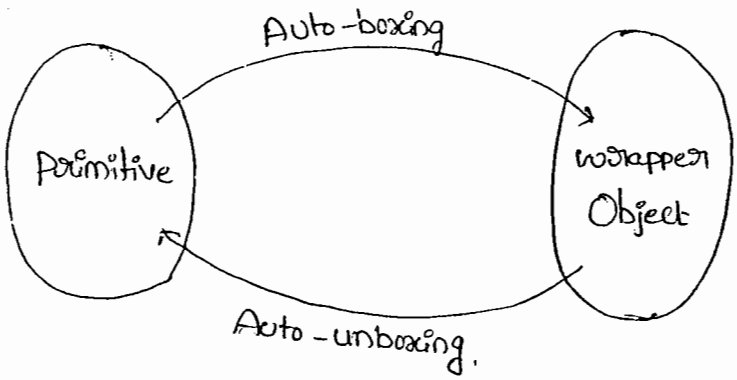
Ex:- ✓ Integer I = 10; [Compiler Converts int to Integer automatically by Autoboxing].

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 implimented by using ValueOf()

Ex②:-

```
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(1):-

```
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.U.B
    S.o.pln(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 interchangeably.

Ex2:-

```
class Test
{
Static Integer I=0;
P.S.V.m(String[] args)
{
  int i = I;
  S.o.pln(i); //0
}
}
```

int i = I.intValue();

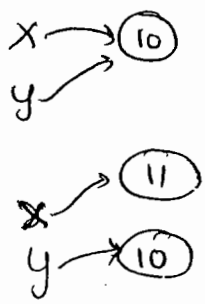
```
class Test
{
Static Integer I;
P.S.V.m(String[] args)
{
  int i = I;
  S.o.pln(i);
}
}
```

→ R.E:- NPE

int i = I.intValue()
↓
null

Ex3:-

```
Integer x = 10;
Integer y = x;
x++;
✓ S.o.pln(x); 11
✓ S.o.pln(y); 10
✓ S.o.pln(x==y); false
```



Note:- because if we want to changes after creating an object, then that new changed object is created with the same reference name.

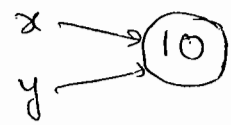
Ex4:-

① Integer X = new Integer(10);
Integer Y = new Integer(10);
S.o.pln(X==Y); false ✓

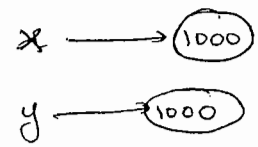
② Integer X = new Integer(10);
Integer Y =10;
S.o.pln(X==Y); false ✓

③ Integer x = 10;
Integer y = 10;
S.o.pln(x == y); true ✓

④ Integer x = 100;
Integer y = 100;
S.o.pln(x == y); true ✓



⑤ Integer x = 1000;
Integer y = 1000;
S.o.pln(x == y); false ✓



Conclusion :-

→ By AutoBoxing if an Object is required to Create Compiler won't Create that object immidiately. 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 Compulsary a new object will be Created.

http://javabynataraj.blogspot.com

Ex:-

① Integer I_1 = 127;
Integer I_2 = 127;
S.o.pln (I_1 == I_2); true

② Integer I_1 = 128;
Integer I_2 = 128;
S.o.pln (I_1 == I_2); false

③ Float f_1 = 10.0f;
Float f_2 = 10.0f;
S.o.pln (f_1 == f_2); false

④ Boolean b_1 = true;
Boolean b_2 = true;
S.o.pln (b_1 == b_2); true.

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

→ 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.opln("widening");
  }
  p.s.v.m1(Integer I)
  {
    S.o.pln("Autoboxing");
  }
}
```

```
p.s.v.m(String[] args)
{
    int x=10;
    m1(x);      o/p:- widening
}
}
```

→ widening (1.0V) dominates Auto-boxing (2.0)

Case(2):-

→ Widening Vs Var-arg():

Ex:-
```
Class Test
{
    p.s.v.m1(long l)
    {
        S.o.pln("widening");
    }
    p.s.v.m1(int... i)
    {
        S.o.pln("Var-arg");
    }
    p.s.v.main(String[] args)
    {
        int x=10;
        m1(x);      o/p:- widening
    }
}
```

→ widening dominates Var-arg().

Case 3:-

→ Auto-boxing Vs Var-arg:-

Ex:-
```
Class Test
{
  p.s.v.m1(Integer I)
  {
    S.o.pln("Autoboxing");
  }
  p.s.v.m1(int... i)
  {
    S.o.pln("var-arg");
  }
  p.s.v.m(String[] args)
  {
    int x=10;
    m1(x);        o/p:- Autoboxing.
  }
}
```

→ In General Var-arg() will get least priority, If no other method matched then only Var-arg() will be Executed.

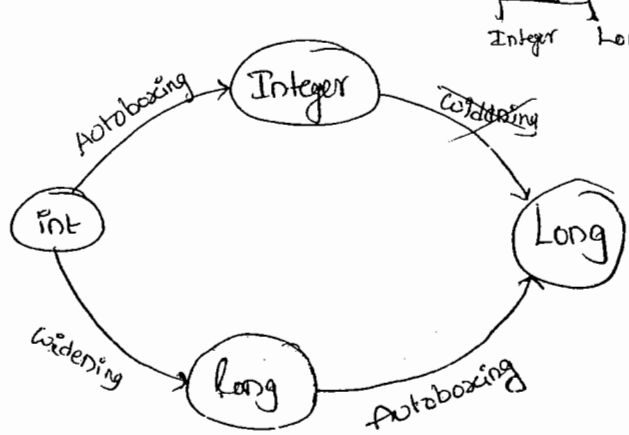
→ While resolveing overloaded methods Compiler will always keeps the precidence in the following order.

(i) Widening

(ii) Auto-boxing

(iii) Var-arg().

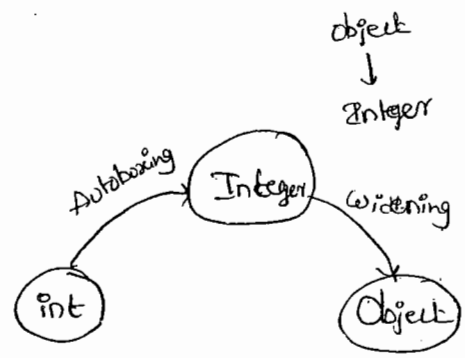
Case 4 :-




```
Class Test
{
  p.s.v.m1(Long l)
  {
    S.o.pln("Long");
  }
  p.s.v.main(String[] args)
  {
    int x=10;
    m1(x);
  }
}
```

C.E:-
m1(java.lang.Long) in Test cannot be applied to (int)

→ widening followed by Auto-boxing is not allowed in java. where as Autoboxing followed by widening is allowed.



Ex:-

```
Class Test
{
  p.s.void m1(Object o)
  {
    S.pln("Object");
  }
  p.s.void.main(String[] args)
  {
    int x=10;
    m1(x); // 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;

310

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 :-

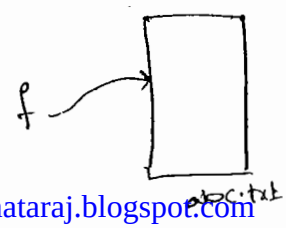
* File f = new File("abc.txt");

→ This line wont Create any physical file, first it will check is their 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.pln(f.exists());  // false
f.createNewFile();
S.o.pln(f.exists());  // true
```



→ A Java file object can represent a directory also.

Ex:-
```
File f = new File("durga123");
S.o.p(f.exists());  false
f.mkdir();
S.o.p(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:- ① Write code to create a file named with abc.txt in current working directory.

```
File f = new File("abc.txt");
f.createNewFile();
```

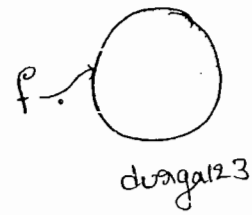
② w.c. to create a directory named with xyz in current working directory.

```
File f = new File("xyz");
f.mkdir();
```

③ w.c. to create a directory named with durga123 in current working directory. in that directory create a file named with abc.txt.

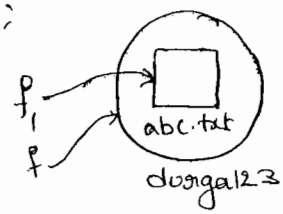
A)

```
file f = new File("durga123");
f.mkdir();
```




```
file f1 = new file("durga123", "abc.txt");
f1.CreateNewFile();
```

(or)




```
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 written returns false without creating new file. If it is not already available then this method returns true after creating new file.

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
{
  p.s.v.m(String[] args) throws Exception
  {
    File f = new file("D:\\durga_classes");
    String[] s = f.list();
    for (String s1 : s)
    {
      S.o.pln(s1);
    }
  }
}
```

(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 overridding. If we want to perform append instead of overridding 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.

(4) flush() :-

→ to give the guarantee that Last character of the data also return to the file.

(5) close() :-

Ex:- demo program for the FileWriter.

```
import java.io.*;
Class FileWriterDemo2
{
  P.S.V.M(String[] args)
  {
    FileWriter fw = new FileWriter("wc.txt", true);   → appending
    fw.write(100); // adding a single character
    fw.write("Durga\n SoftwareSolutions");
    Char[] ch = {'a','b','c'};
    fw.write('\n');
    fw.write(ch);
    fw.write('\n');
    fw.flush();
    fw.close();
  }
}
```

O/P :- durga 100 → d
SoftwareSolutions
abc

(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);
  S.o.pln(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);
  }
  S.o.pln("***----------");
  FileReader fr1 = new FileReader(f);
  int i = fr1.read();
  while (i != -1)
  {
    S.o.pln((Char) i);
    i = fr1.read;
  }
}
}
```

- Usage of FileWriter & FileReader is not recommended because :-

1) While writing data by filewriter we have to insert line seperators manually which is a bigger headache to the programmer.

2) By using fileReader we can read data character by character which is not convineiant ~~waste~~ 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.

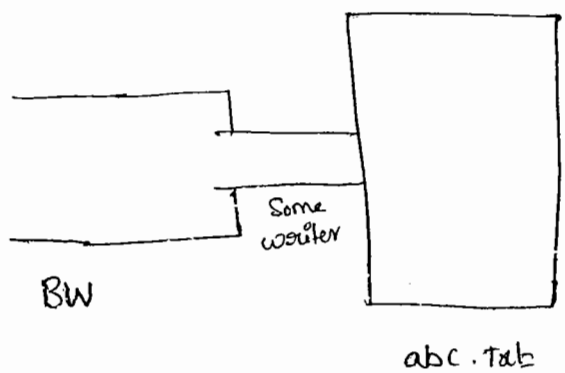
Constructors :-

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.



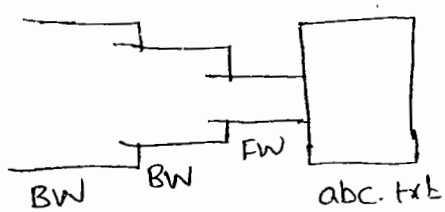
Q) which of the following are Valid.

✗ 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 ch)

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 Seperate method in BufferedWriter.

A) ①

① writeing data to the file.

② flushing the Stream.

③ closeing the Stream.

④ inserting a line seperator.

Ex:-

```
import java.io.*;
class BufferedWriterDemo
{
    P.S.V.m(String[] args) throws IOException
    {
        File f = new File("wc.txt");
        FileWriter fw = new FileWriter(f);
        BufferedWriter bw = new BufferedWrite(fw);
        bw.write(100);
        bw.newLine();
        char[] ch1 = {'a','b','c','d'};
        bw.write(ch1);
        bw.newLine();
        bw.write("durga");
        bw.newLine();
        bw.write("Software Solutions");
        bw.flush();
        bw.close();
    }
}
```

O/P:-

```
d
abcd
durga
SoftwareSolutions
```

wc.txt

Note:- When ever we are closeing BufferedWriter automatically underlaying writers will be closed

BW.close(); ✓	fw.close(); ×	fw.close(); bw.close(); ✗

(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 buffer size);

Note :-

* BufferedReader can't Communicate directly with the file compulsary it should Communicate via some reader object.

Important methods :-

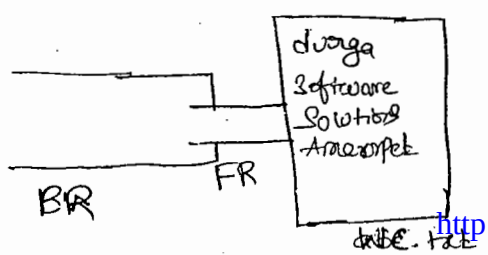
① int read();

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
{
  p.s.v.m(String[] args) throws Exception
  {
    FileReader fr = new FileReader("wc.txt");
    BufferedReader br = new BufferedReader(fr);
    String line = br.readLine();
    While(line != null)
    {
      S.o.pln(line);
      line = br.readLine();
    }
    br.close();
  }
}
```

O/P:- durga
Software
Solutions
-Ameerpet

Note:-

* when ever your closing BufferedReader underlying Readers will be closed.

(v) PrintWriter() :-

* This is the most enhanced writer to write a character data to file. By using FileWriter & BufferedWriter we can write only character data but by using PrintWriter we can write any primitive data types to the file.

Constructors:-

1. PrintWriter pw = new PrintWriter(String name);
2. PrintWriter pw = new PrintWriter(File f);
3. PrintWriter pw = new PrintWriter(Writer w);

Methods:-

1. write(int ch)	print(char ch)	println(char ch)
2. write(char[] ch)	print(int i)	println(int i)
3. write(String s)	print(long l)	println(long l)
4. flush()	print(double d)	println(double d)
5. close()	print(String s)	println(String s)
	print(char[] ch)	println(char[] ch)
	⋮	⋮

Ex:-

```
import java.io.*;
Class PrintWriterDemo1
{
  p.s.v.m(String[] args) throws IOException
  {
```

```
FileWriter fw = new FileWriter("cc.txt");
PrintWriter pw = new PrintWriter(fw);
pw.write(100); // d
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
```

cc.txt

Q:- what is the diff. b/w the following

(a) pw.write(100);

(b) pw.print(100);

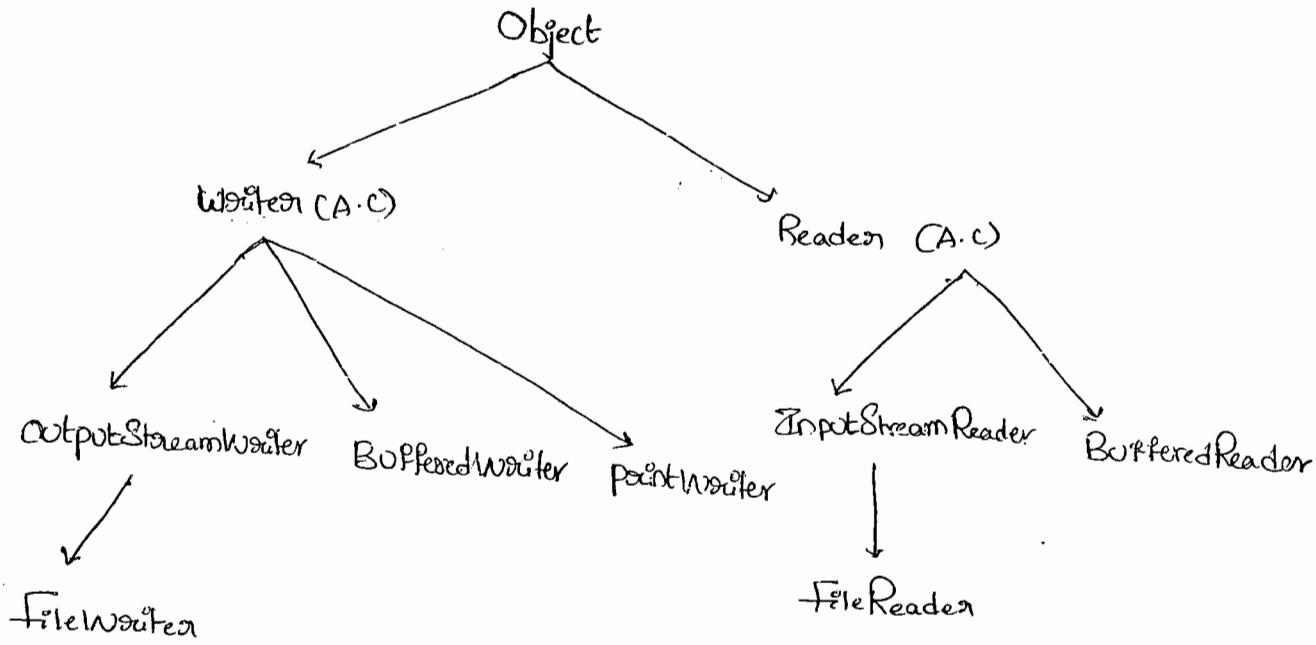
Ans:-

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

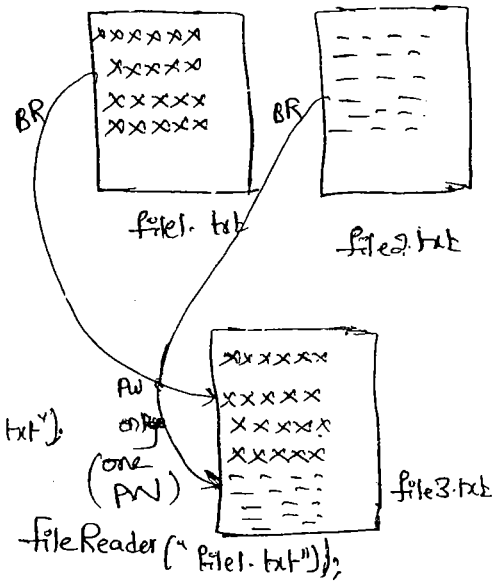


Note:-

* 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 3rd file. 321

file3.txt = file1.txt + file2.txt




```
import java.io.*;
Class FileMerger
{
  p.s.v.m (String[] args) throws Exception
  {
    PrintWriter pw = new PrintWriter("output.txt");
    BufferedReader br = new BufferedReader(new FileReader("file1.txt"));
    String line = br.readLine();
    while (line != null)
    {
      pw.println(line);
      line = br.readLine();
    }
    br = new BufferedReader(new FileReader("file2.txt"));
    line = br.readline();
    while (line != null)
    {
      pw.println(line);
      line = br.readLine();
    }
    pw.flush();
    br.close();
    pw.close();
  }
}
```

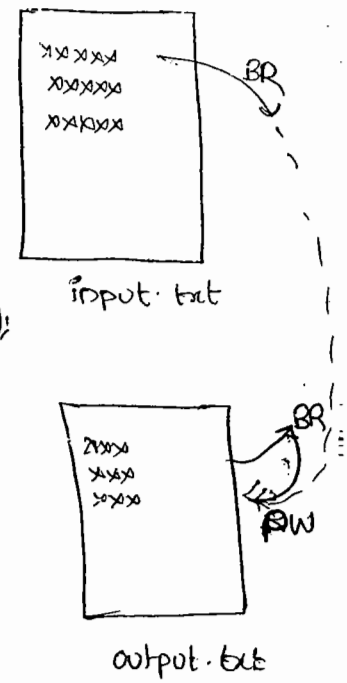
Ex2:-

w.a.p to merge data from 2 files into a 3rd file but merging should be done line by line alternatively.

```
import java.io.*;
Class FileMerger2
{
  p.s.v.m(String[] args) throws IOException
  {
    PrintWriter pw = new PrintWriter("output.txt");
    BufferedReader br1 = new BufferedReader(new FileReader("file1.txt"));
    BufferedReader br2 = new BufferedReader(new fileReader("file2.txt"));
    String line1 = br1.readLine();
    String line2 = br2.readLine();
    while((line1 != null) | (line2 != null))
    {
      if(line1 != null)
      {
        pw.println(line1);
        line1=br1.readLine();
      }
      if(line2 != null)
      {
        pw.println(line2);
        line2 = br2.readLine();
      }
    }
  pw.flush();          pw.close();
  br1.close();
  br2.close();
  }
}
```

3).
W.a.p to merge data from two files into a 3rd file But merging should be done para by para. assume that there is a Blank Line B/w Every 2 paras?

4).
W.a.p to delete duplicates from the given i/p file?




```
import java.io.*;
class DuplicateEliminator
{
 p.s.v.m(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;
  BR 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();
  }
```

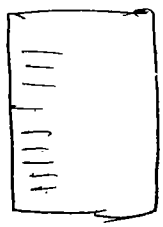
```
if (available == false)
 {
   pw.println(line);
   pw.flush();
 }
 line = br1.readLine();
}
pw.flush();
br1.close();
br2.close();
pw.close();
}
}
```

(5) W.a.p to perform file Extraction (result.txt = total.txt - delete.txt)

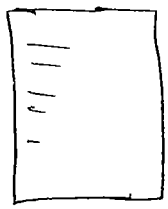
```
import java.io.*;
Class FileExtraction
{
  p.s.v.m (String[] args) throws Exceptions
  {
    BufferedReader br1 = new BufferedReader(new FileReader
                                ("mobile.txt"));
    PrintWriter pw = new PrintWriter("output.txt");
    String line = br1.readLine();
    while (line != null)
    {
      boolean available = false;
      BR br2 = new BR(new FR("delete.txt"));
```

durgajobs.com

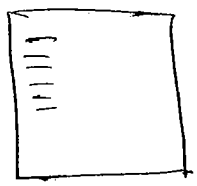
```
String target = br2.readLine();
while(target != null)
{
  if(line.equals(target))
  {
    available = true;
    break;
  }
  target = br2.readLine();
}
if(available == false)
{
  pw.println(line);
}
line = br1.readline();
}
pw.flush();
br1.close();
br2.close();
pw.close();
}
}
```



total.txt



delete.txt



result.txt

Profession etc

heretic = a person who holds Controversial beliefs, especially Contrary to religion,

prolonged argument.

opposite

324

Serialization

① Introduction

② Object Graphs in Serialization

③ Customized Serialization

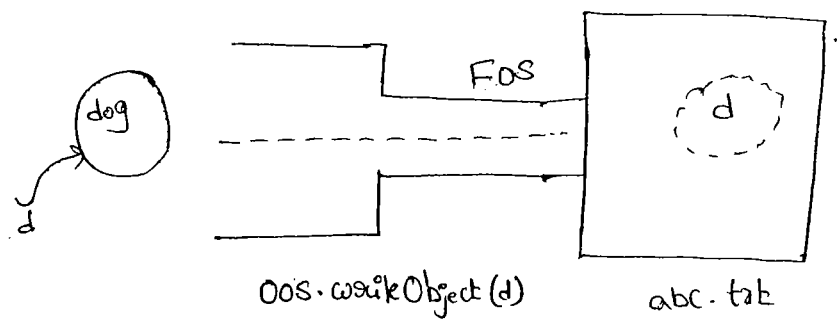
④ Serialization w.r.t Inheritance.

Serilization :-

→ 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.

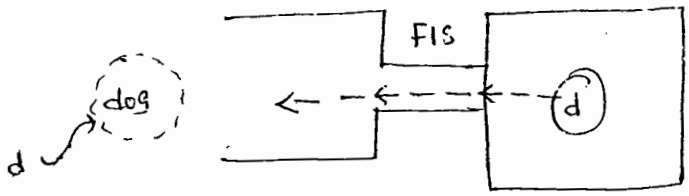


DeSerialization :-

→ 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 file Supported form to java Supported form.

→ By using FileInputStream and ObjectInputStream classes we Can achieve De-Serialization.



OIS.readObject(d)

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();
```

Serilization

```
    FileOutputStream fos = new FileOutputStream("abc.txt");
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(d1);
```

```
FileInputStream fis = new FIS("abc.tab");
OIS ois = new OIS(fis);
Dog d2 = (Dog) ois.readObject();
S.o.pln (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 <u>java.io</u> package and doesn't contain any methods, it is a marker interface.

→ If we are trying to serialize a non-Serializable object we will get Run time exception Saying <u>NotSerializableException</u>.

<u>transient Keyword</u> :-

→ At the time of Serialization if we dont want to Serialize the value of a particular variable to meet the Security Constraint we have to declare those variables with "<u>transient</u>" keyword.

→ At the time of Serialization JVM ignores original value of transisent 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 transistent there is no impact.

transisent Vs final :-

→ final variables will be participated into Serialization directly by their values Hence declaring a final variable with transistent there is no impact.

Summary :-

declaration	o/p
① int i = 10 int j = 20	10 ----- 20
② transient int i = 10; int j = 20	0 ----- 20
③ transient final int i = 10; transient int j = 20;	10 ----- 0
④ transient int i = 10; transient static int j = 20;	0 ---- 20

Object Graph in Serialization :-

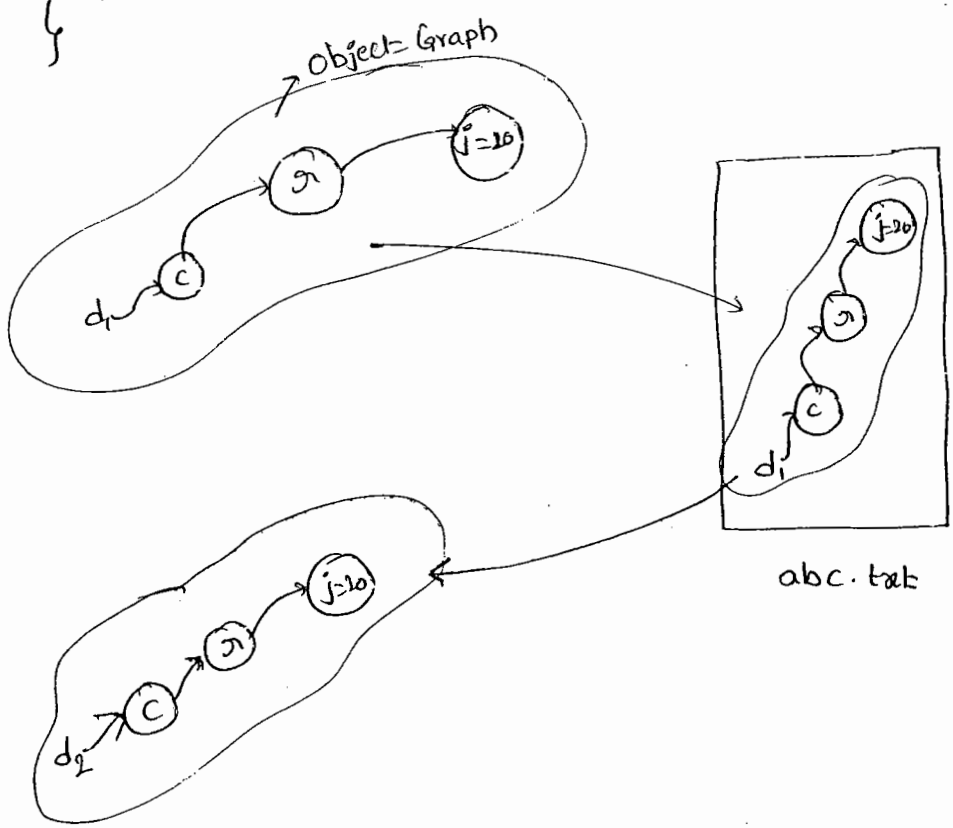
→ when ever we are trying to Serialize an object the Set of all objects which are reachable from that object will be Serialized automatically this group of objects is called "Object Graph".

→ In Object Graph every object should be Serializable otherwise we will get "Not-Serializable-Exception".

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 d1 = (Dog) ois.readObject();
S.o.pln(d1.c.r.j); //20
}
}
```



→ In the above prog. when ever we are Serializing a Dog object automatically Cat & Rat objects will be Serialized. because these are the part of object graph of dog.

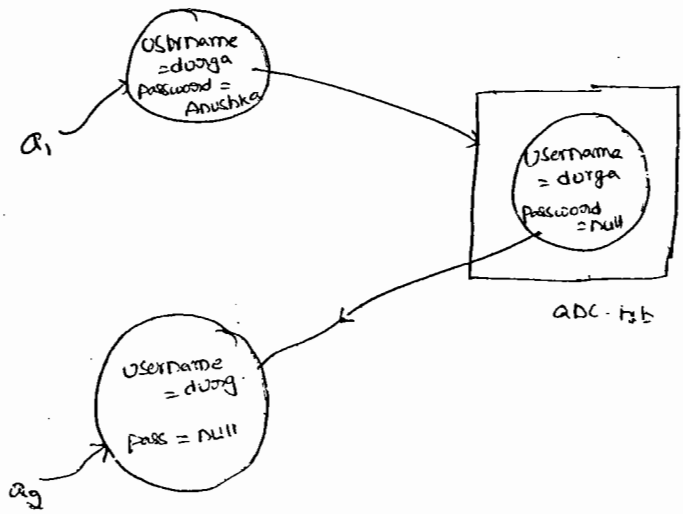
→ Among dog, Cat & Rat if atleast one class is not Serializable Then we will get NotSeriableException.

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 SerializeDemo3
{
  P.S.V.m(String[] args) throws Exception
  {
    Account a1 = new Account();
    S.o.pln(a1.username + "-----" a.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.pln(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.

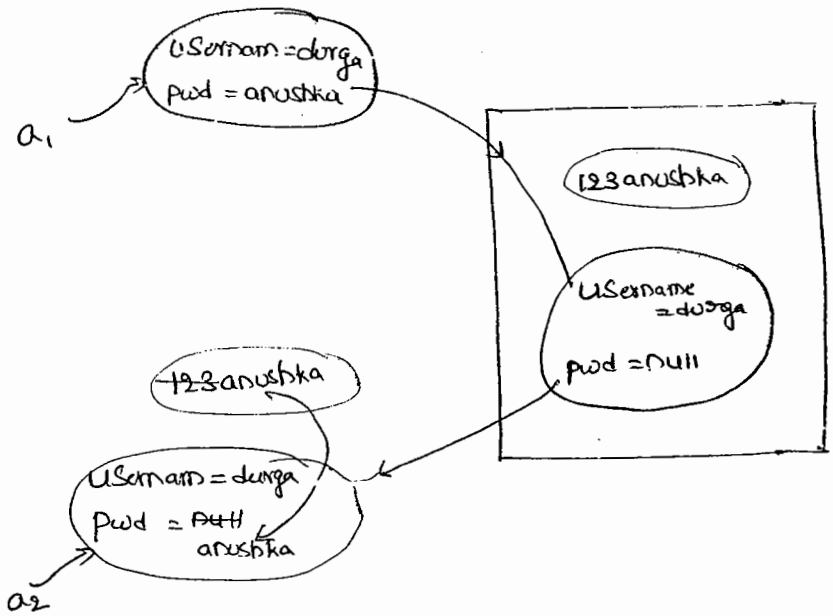
→ The above 2 methods we have to define in the Corresponding Class of Serialized Object.

ex:-

```
import java.io.*;
class Account implements Serializable
{
  String username = "durga";
  transient String pwd = "anushka";
  private void writeObject(ObjectOutputStream os) throws Exception
  {
    os.defaultWriteObject();
    String epwd = (String)is.readObject();
    pwd = epwd.substring(3);
  }
}
class CustSerializeDemo1
{
  p.s.v.m(String[] args) throws Exception
  {
    Account a1 = new Account();
    S.o.pln(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.pln(a2.username + "----" + a2.pwd);
    }
}
```

~~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
{
    int x=10;
}
class Dog extends Animal
{
    int y=20;
}
```

→ we can Serialize dog object eventhough dog class doesn't implement Serializable interface Explicitly. because it's parent class Animal is Serializable.

Case 2:-

→ Eventhough 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 checkes is any parent class is non-Serializable or not, JVM Creates a Seperate Object for every non-Serializable parent & share its instance variables to the Current Object.

→ for this JVM always calls no arguement Constructor of the non-Serializable parent. If the non-Serializable parent doesn't have no arrguement Constructor then we will get RuntimeException.

Ex:-

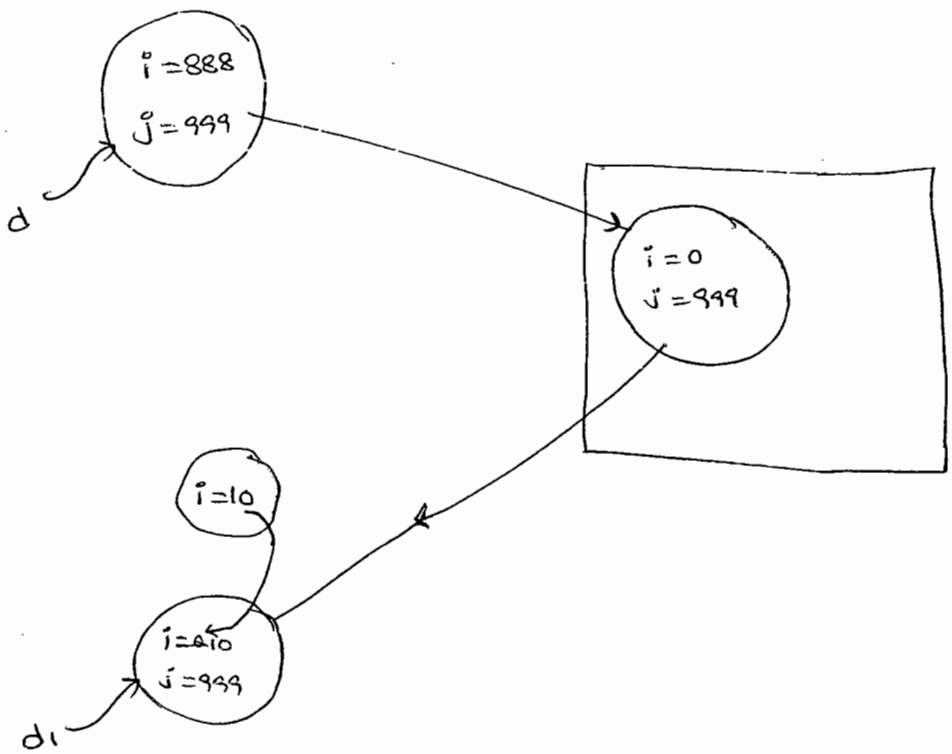
```
import java.io.*;
Class Animal
{
  int i=10;
  Animal()
  {
    S.o.pln("Animal Constructor Called");
}  }
```

```
Class Dog extends Animal implements Serializable
{
    int j=20;
    Dog()
    {
        S.o.pln("Dog Constructor Called");
    }
}
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.pln("Deserialization Started");
        FileInputStream fis = new FileInputStream("abc.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Dog d1 = (Dog)ois.readObject();
        S.o.pln(d1.i+"---"+d1.j);
    }
}
```

O/P :- Animal Constructor Called

Dog " "

Deserialization started

* Animal Constructor Called

10 - - - - 999