21/02/11

Language fundamentals

1) Identifier :-

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

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 not recommended to take morethan 15 length (>15).

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

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

Ex:-

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

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

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

✓ ① Java2Share
✗ ② 4shared
✗ ③ all@hands
✓ ④ total_no_of_Students
✓ ⑤ _$_
✗ ⑥ total#
✗ ⑦ int
✓ ⑧ Integer

Primitive data types (8)

- Numeric datatypes (To represent numbers)
  - 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)

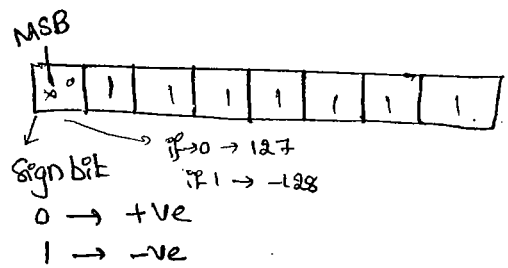
① Byte :-

Size = 8-bits (or) 1 Byte

Max_value = 127

Min_value = -128

Range = -128 to +127



→ 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 :-

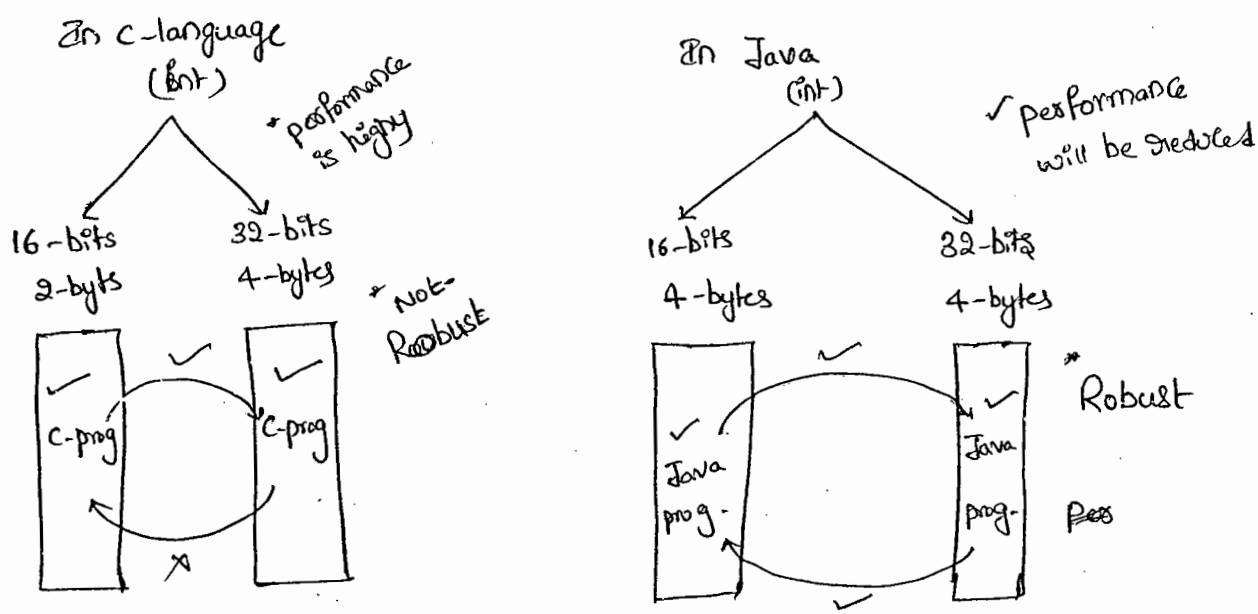
→ In C language the size of int is varied from platform to platform for 16-bit processors it is 2-bytes but for 32-bit processors it is 4-bytes

* 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~~ 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.

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

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

2)

Ex(2):-

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

Size = 8 bytes

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

Note:-

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

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

Floating Point data types :-

floating point data-types

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

Boolean data type :-

Size : Not Applicable (virtual machine dependent)

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

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

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

Size : 2-bytes

Range : 0 to 65535

Summary of primitive data types :-

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

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

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

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

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

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

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

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

Ex②:-

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

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

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

O/P: Hi

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

O/P:- Hi

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

O/P:- Hello.

Char Literals :-

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

రాష్ట్రాలు :

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';

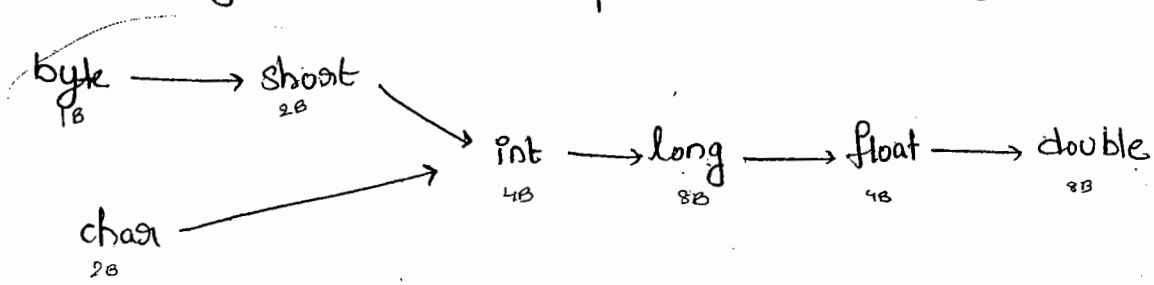
✓ 5) char ch = 'a';

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:- ✗ int[6] a;

(b) 2D Array declaration:-

✓ 1) int[][] a;

✓ 2) int [][]a;

✓ 3) int a[][];

✓ 4) int[] a[];

✓ 5) int[] []a;

✓ 6) int []a[];

c) 3D - Array declarations:-

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

Q) Which of the following are valid declarations.

✓ 1) int[] a,b; a→1 b→1

✓ 2) int[] a[],b; a→2 b→1

✓ 3) int[] []a,b; a→2 b→2

✓ 4) int[] []a,b[]; a→2 b→3

✗ 5) int[] []a,[]b; a→2 b→3 C.E:-

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

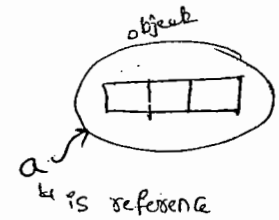
Ex:- int[] []a, []b;

Allowed — not allowed

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;

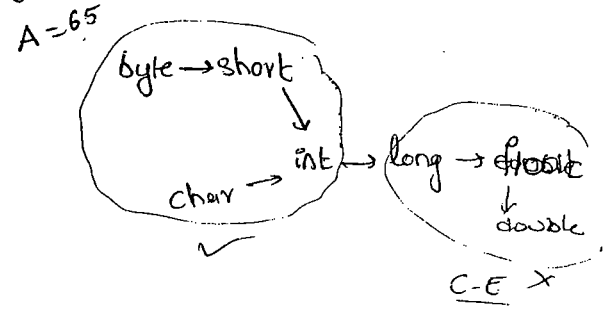
✓ int[] a = new int[b];

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.

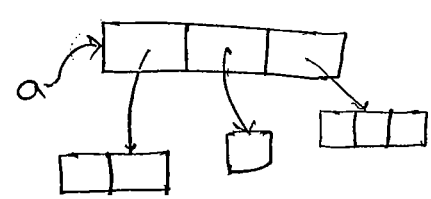
→ The main advantage of this approach is memory utilization will be improved.

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

a[0] = new int[2];

a[1] = new int[1];

a[2] = new int[3];

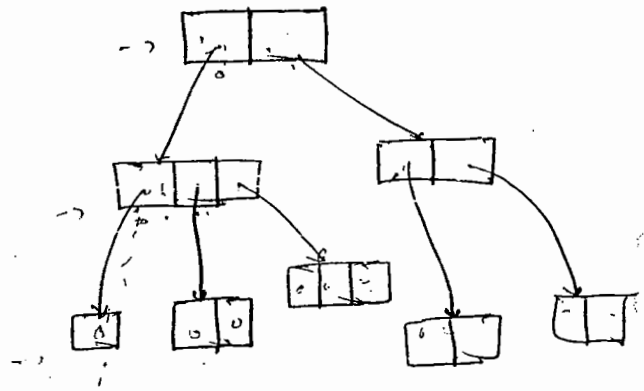


Note:-

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];

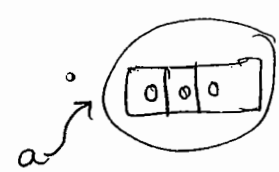
Array Initialization:-

→ 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

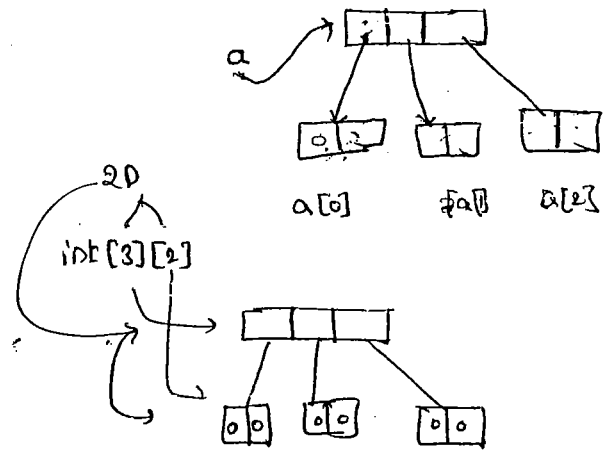


Note:- Whenever we are trying to print any object reference internally toString() will be call which is implemented as follows.

Classname @ hexadecimal_String_of_hashCode.

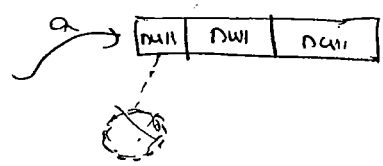
Ex (2):-

```
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):-

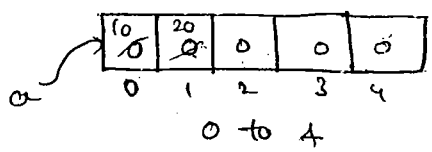
```
int[][] a = new int[3][];
S.o.pln(a);        [[I@-----
S.o.pln(a[0]);     null
S.o.pln(a[0][0]);  R.E:- NPE
```



→ 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

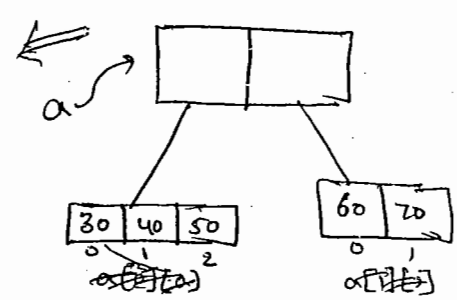
Ex(2):- `char[] ch = {'a', 'e', 'i', 'o', 'u'};`

`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}};`

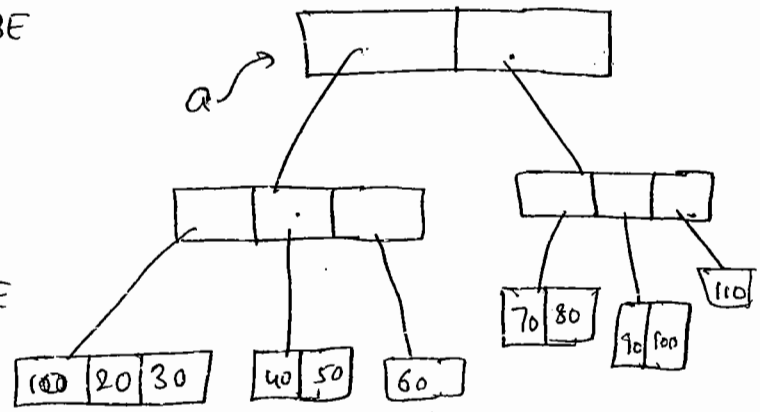


→ We can Extend this Short Cut Even for 3D array also

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

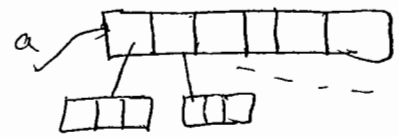
location : java.lang.String.

→ 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. (only onetime) (not future).

→ 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 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 no longer Annonymous.

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) :-

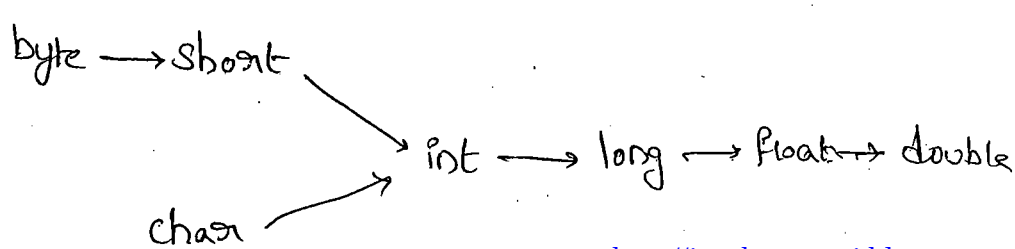
→ for the primitive type arrays as Array elements we can provide any type which can be promoted to declare type.

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.

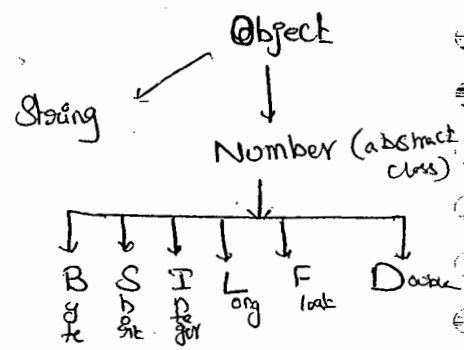
② Object[] a = new Object[10];

✓ 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:- Incompatible type
found : char[]
required : int[]

Q) Which of the following promotions are valid.

✓ ① char → int

✗ ② char[] → int[]

✓ ③ int → long

✗ ④ int[] → long[]

✗ ⑤ long → int

✗ ⑥ long[] → double[]

✓ ⑦ String (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):-

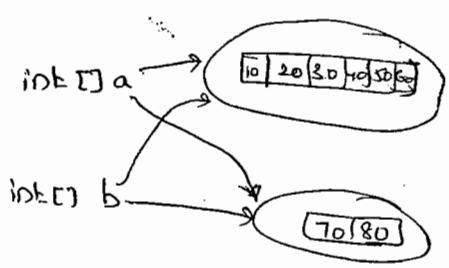
→ When even we are assigning one array to another array only reference variables will be reassign but not underlying elements. Hence types must be matched but not sized.

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

int[] b = {70, 80};

✓ ① a = b;

✓ ② b = a;



Eg(2):

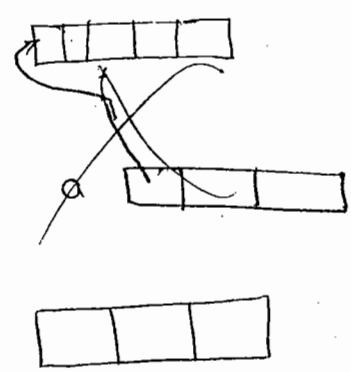
int[][] a = new int[3][2];

a[0] = new int[5];

a[1] = new int[4];

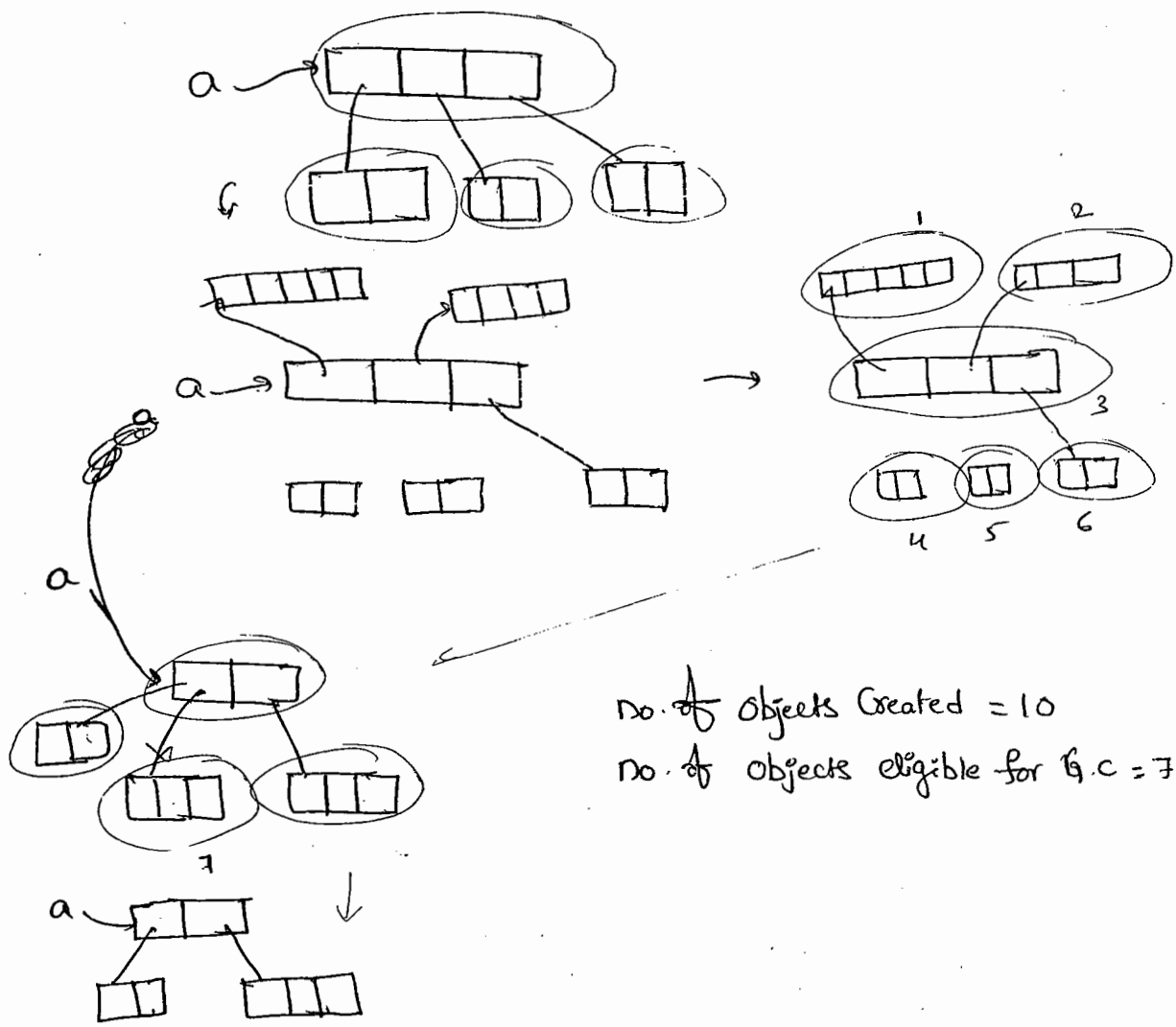
a = new int[2][3];

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[]

22

Types of Variables

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

(i) primitive variables
(ii) reference variables

(i) Primitive Variables:-

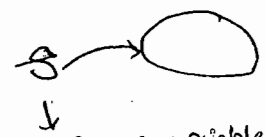
→ Can be used to represent primitive values

ex:- int x = 10;

(ii) reference variables:-

→ Can be used to refer Objects

ex:- Student s = new Student();



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"
```

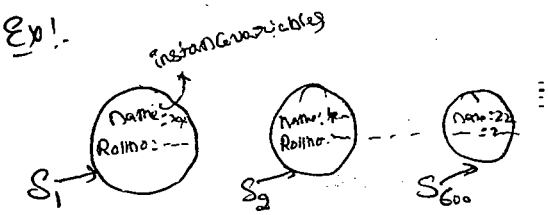
```
Test t = new Test();
S.o.pln(t.x); 10
}
public void m1()
{
  S.o.pln(x); 10
}
}
```

→ 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:-



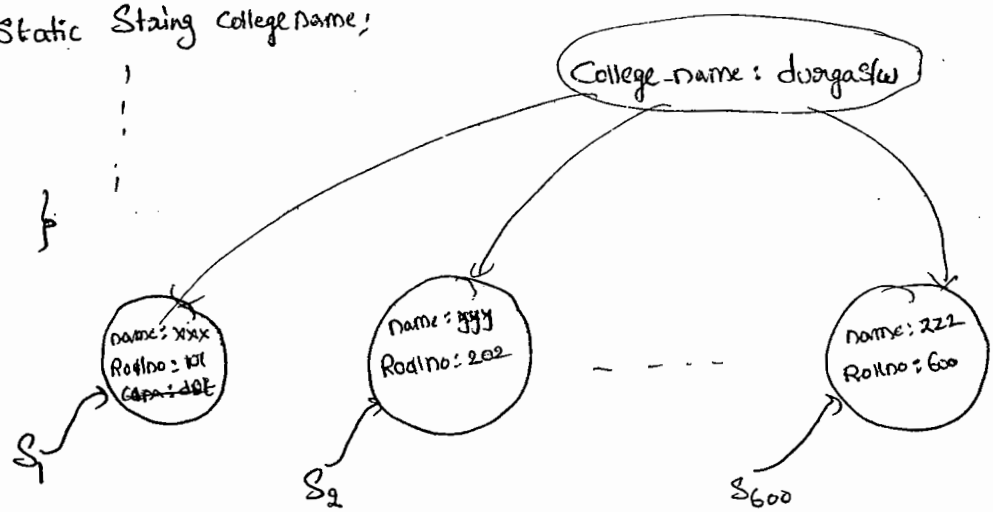
Students objects, in that name, Rollnos are instance variables, Bcz, these values are varied from object to object.

→ 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.

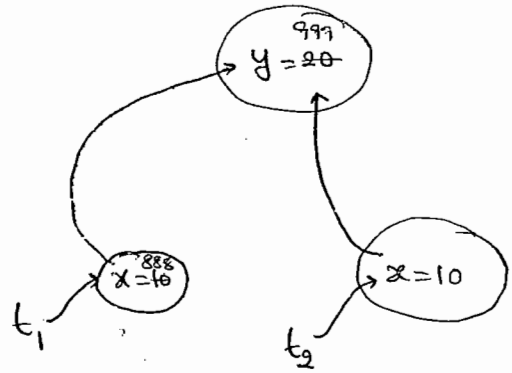
eg:-

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

→ 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
}
}
```

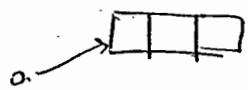
Instance level :-

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

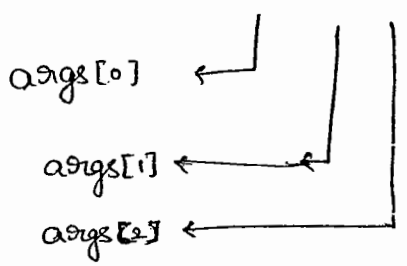
Command Line Arguements

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 arrguement 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.