Durgasoft SCJP Notes

Part-1

21/02/11

Language fundamentals

1) Identifiers :-

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

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

↳ variable name

* Rules to define identifiers :-

1) The only allowed characters in Java identifier are :

✓ (a to z, A to Z, 0 to 9, _, $)

→ If we are using any other characters we will get Compiletime Error.

Eg:-
- ✓ all_member
- ✗ all#
- ✓ _$_$
- ✗ 098$_10

2) identifier can't starts with digit. Eg:- ✗ 123total
✓ total123.

3). Java identifiers are CaseSensitive.

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

} We can differentiate w.r.t Case.

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

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

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

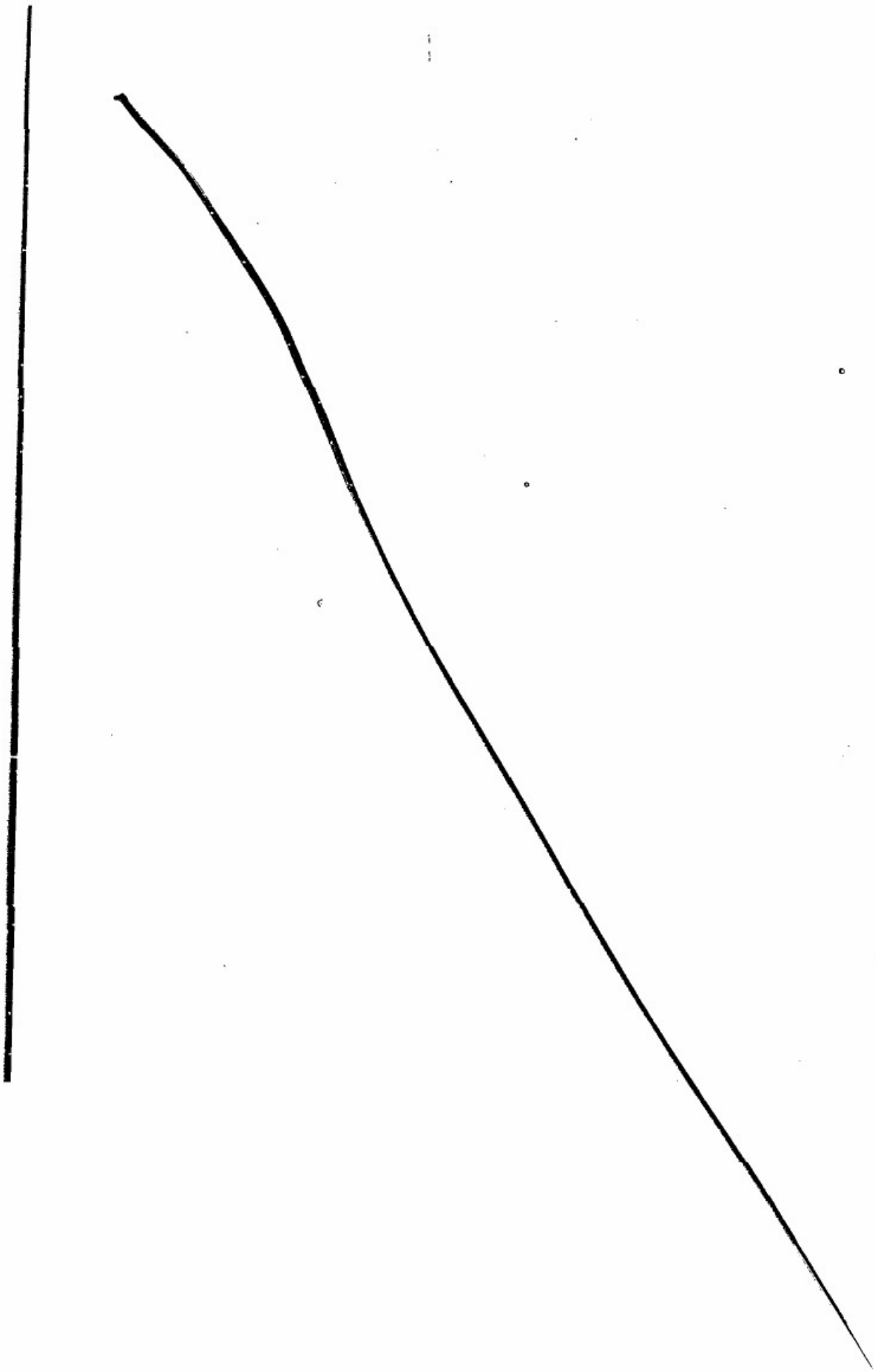
Eg:-

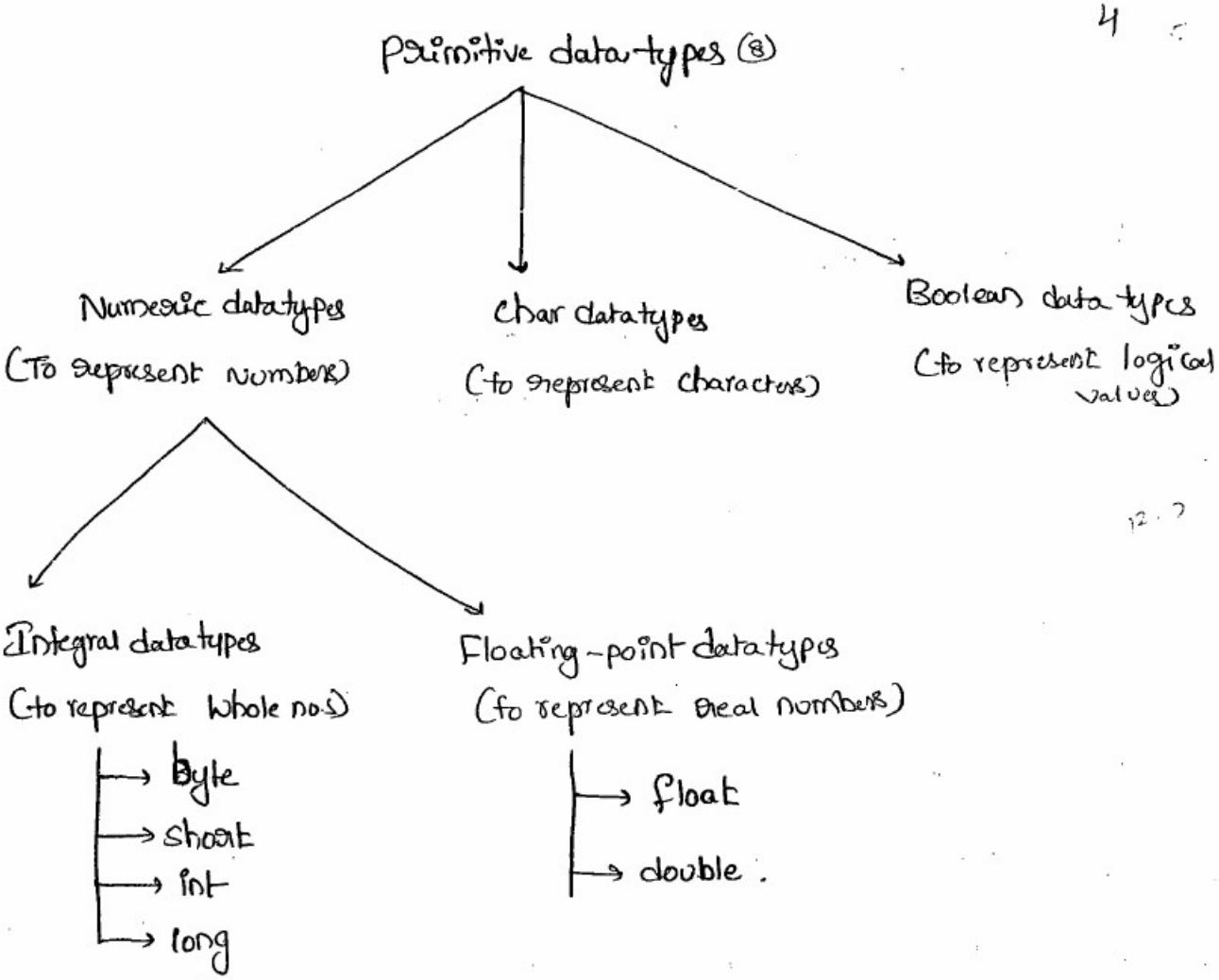
```
Class Test
{
int String = 10;
S.o.pln(String); 10
}
```

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

Q) Which of the following are valid Java identifiers?

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





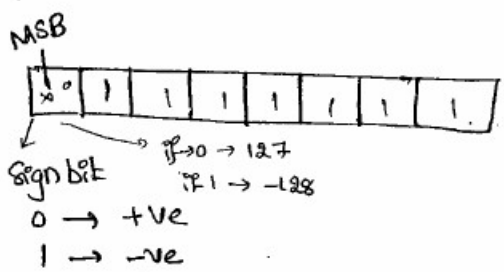
① Byte :-

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 foorm.

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: ~~String~~ 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 out dated & hence Corresponding Short datatype is also out dated.

③ int :-

→ The most Commonly used datatype is "int"

Size : 4-bytes

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

[-2147483648 to 2147483647)

Note :-

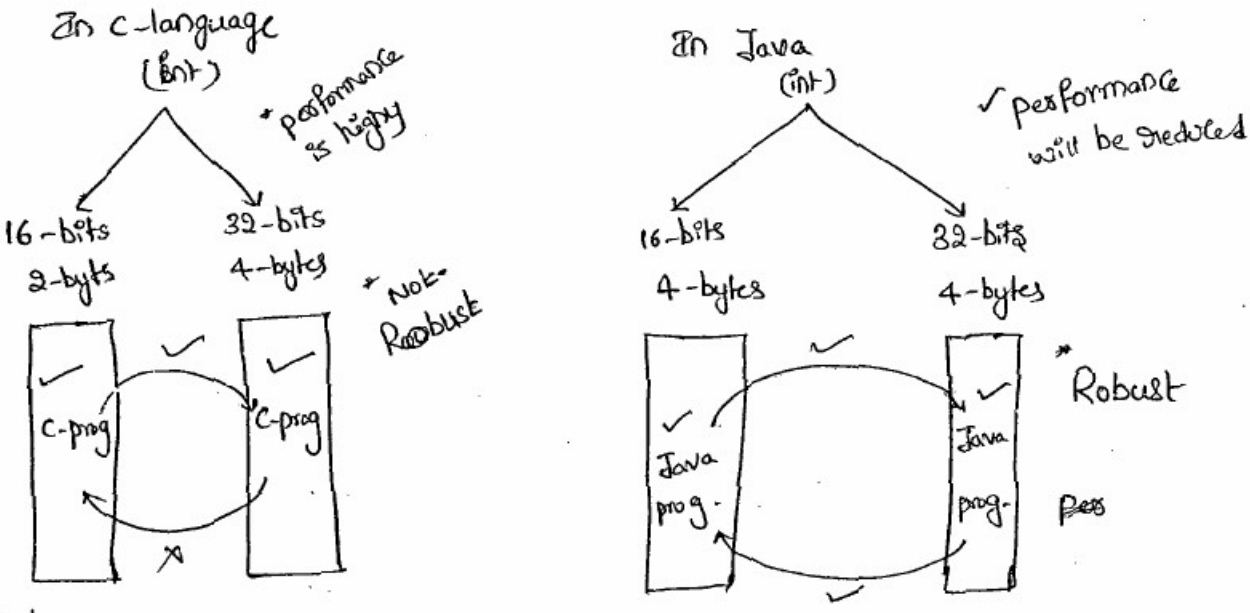
→ In C language the size of int is varied from platform to platform for 16-bit processors it is 2-bytes but for 32-bit processors it is 4-bytes

* The main advantage of this approach is Read & write operation we can perform very efficiently and performance will be improved. But the main

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

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

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



13/08/11

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 x 60 x 60 x 24 x 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

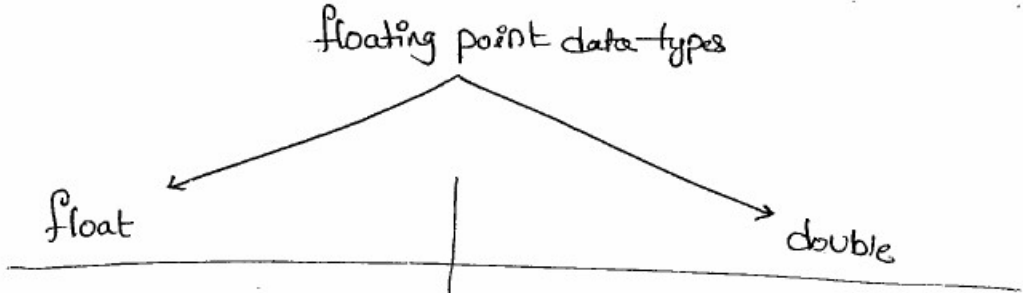
Range = -2^{63} to $2^{63}-1$

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 :-



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 follows Single precision	4) double follows double precision

Boolean data type :-

Size : Not Applicable (Virtual machine dependent)

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

Q) Which of the following boolean declarations are valid

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

✓ 2) boolean b = true;

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

X 4) boolean b = "false" C.E:- incompatible type
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 X

C++ ✓

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

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

in Java X

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

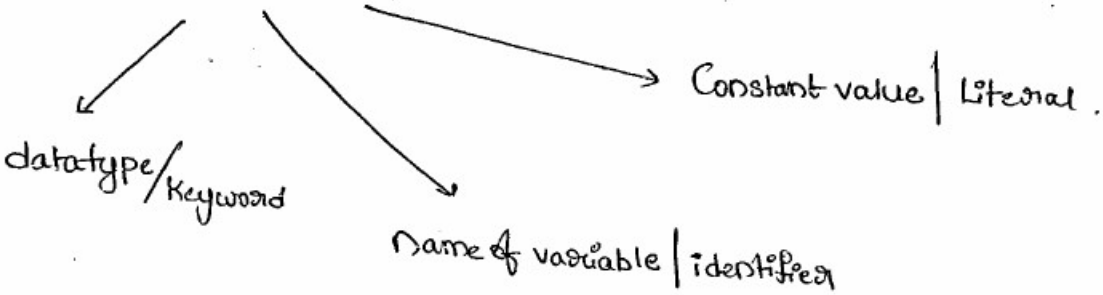
Summary of primitive data types :-

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;`



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

$(10)_8 = (8)_{10}$
$0\times8^0 + 1\times8^1 = 8$

$(10)_{16} = (?)_{10}$
$0\times16^0 + 1\times16^1 = 16$

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; C.E:- P.L.P
found : double
required : float

✓ float f = 123.456f;

✓ double d = 123.456;

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

Eg: ✓ 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.

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

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

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

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

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

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.p(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 `\u xxxx` 4-digit hexa decimal no.

Eg:- ✓ 1) char ch = '\u0061';
S.o.p(ch); a

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

✓ 3) char ch = '\uface';

✗ 4) char ch = 'li beaf';

4. → Every escape character is a char literal

Eg:- ✓ 1) char ch = '\n';

✓ 2) char ch = '\t';

✗ 3) char ch = '\l';

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

Q) Which of the following are valid char declarations.

✓ 1) char ch = 0xbeaf;

✗ 2) Char ch = \u beaf; because ' '

✗ 3) char ch = -10;

✗ 4) char ch = '\x';

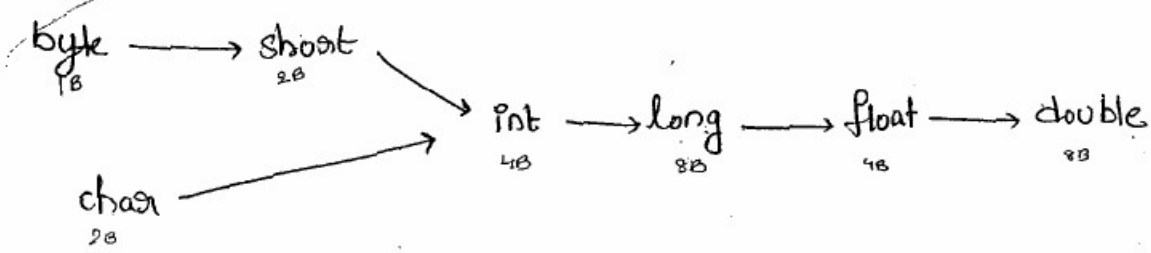
✓ 5) char ch = 'a';

String Literals :-

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

Ex:- String s = "Java";

→ The following promotions will be performed automatically by the Compiler



Arrays (20)

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

Array:-

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

→ The main advantage of array is we can represent multiple values under the same name. So, that readability of the code improved.

→ But the main limitation of array is once we created an array there is no chance of increasing/decreasing size based on our requirement. Hence memory point of view arrays concept is not recommended to use.

→ we can resolve this problem by using Collections.

1) Array declarations:-

(a) Single dimensional Array declaration:-

✓ 1) int[] a;

✓ 2) int a[];

✓ 3) int []a;

→ 1st one 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 dimension before the variable it is possible only for the first variable.

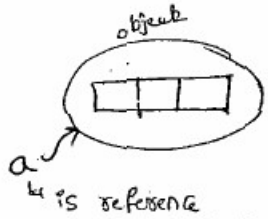
Ex:- `int[] []a, []b;`

Allowed — not allowed;

2) Array Construction:-

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

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



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

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

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

Ex:- `int[] a = new int[];` ✗ C.E:-

`int[] a = new int[3];` ✓

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

Ex:- `int[] a = new int[0];` ✓

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

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

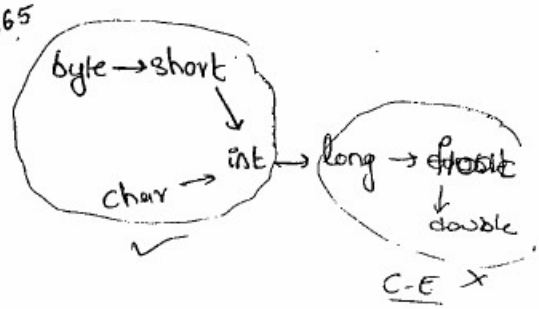
→ To Specify array size the allowed datatypes are byte, short, int char, If we are using any other type we will get C.E.

Ex: ① int[] a = new int['a']; a=97 A=65

② byte b = 10;
int[] a = new int[b];

③ short s = 20;
int[] a = new int[s];

✗ int[] a = new int[10L];
✗ int[] a = new int[10.5];



Note:-

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

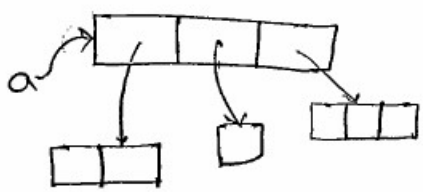
Creation of 2D-Arrays:-

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

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

Ex:-

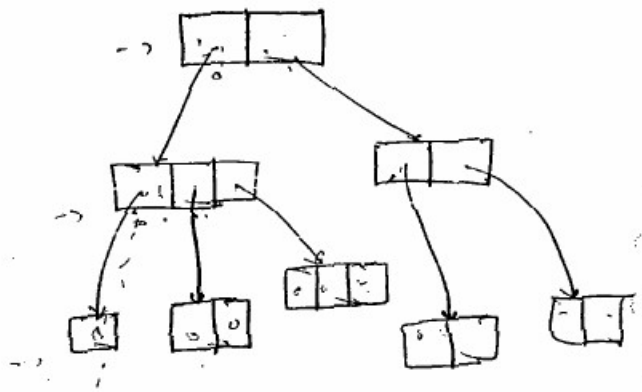
```
int[][] a = new int[3][];
a[0] = new int[2];
a[1] = new int[1];
a[2] = new int[3];
```



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][];`

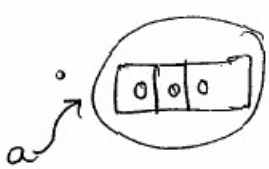
✗ ⑦ `int[][][] a = new int[3][][5];`

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
```

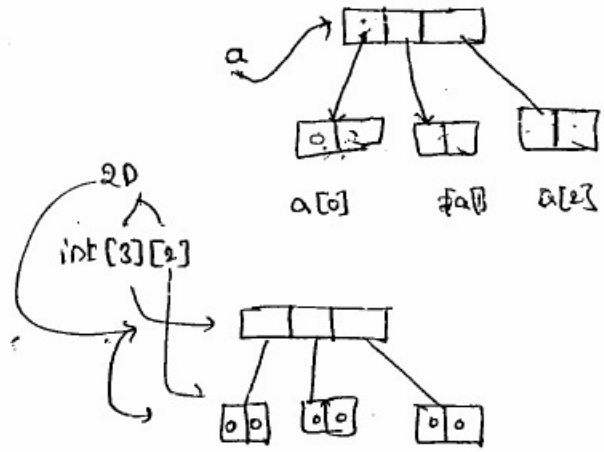


Note:- Whenever we are trying to print any object reference internally toString() will be call which is implemented as follows.

classname @ hexadecimal_String-of-hashCode.

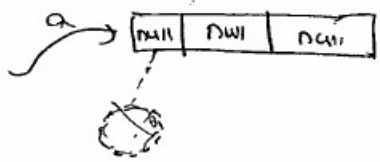
Ex(2):-

```
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):-

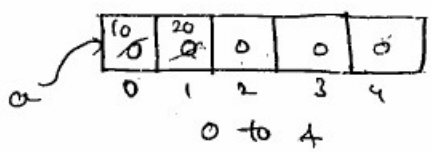
```
int[][] a = new int[3][];
S.o.pln(a);  [[I@-----
S.o.pln(a[0]); null
S.o.pln(a[0][0]); R.E!. NPE
```



→ Once we created an array Every element by default initialized with default value. 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

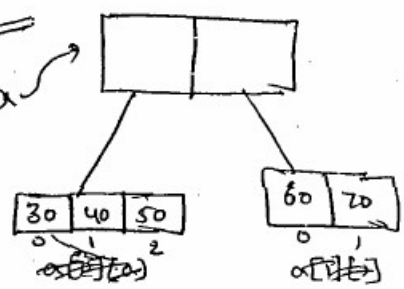
Ex(2) :- `char[] ch = {'a', 'e', 'i', 'o', 'u'};`

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

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

Ex(3) :-

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



→ We can Extend this Shortcut Even for 3D array also

Ex :-

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

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

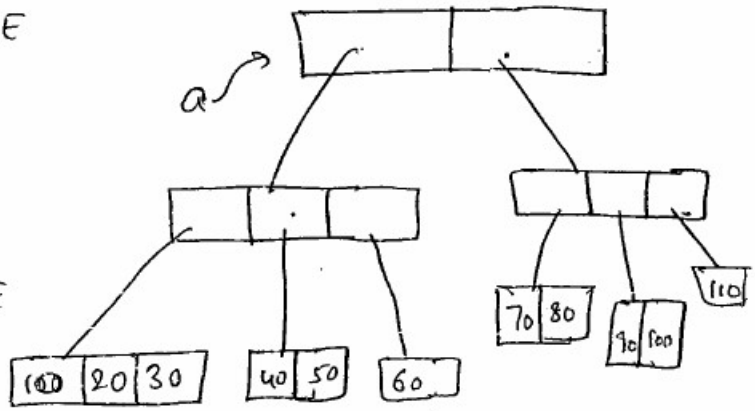
S.o.pln(a[3][1][2]); RE:- AIOBE

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

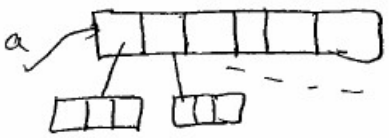
location : java.lang.String.

→ 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 one time)

→ We can create Annonymous Array as follows.

`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 Annonymous 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) :-

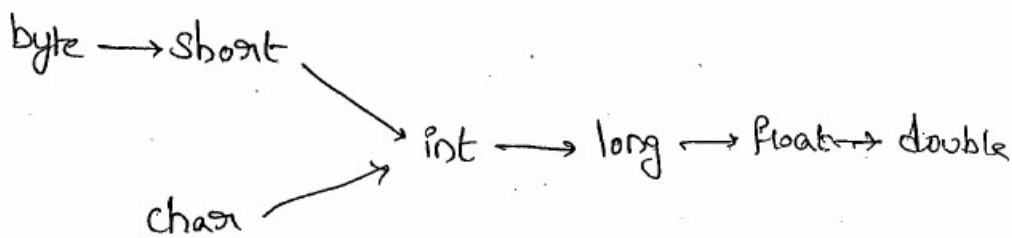
→ for the primitive type arrays as Array elements we can provide any type which can be promoted to declare type.

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:- Incompatible types
found : String
required : Number.

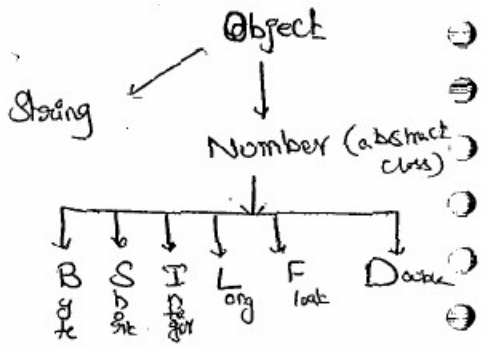
② Object[] a = new Object[10];

✓ a[0] = new Object();

✓ a[1] = new Integer(10);

✓ a[2] = new Double(10.5);

✓ a[3] = new String("durga");



Case(3):-

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

Eg1:- ① Number[] n = new Number[10];

✓ n[0] = new Integer(10);

✗ n[1] = new Number();

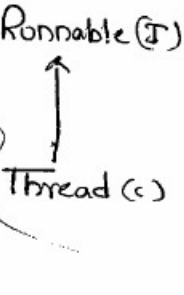
Case 4:-

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

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

r[0] = new Thread(); ✓

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



Note:-

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

Array Variable Assignment :-

Case(1):-

→ Element level promotions are not applicable at array level

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

① `int[] a = {10,20,30,40};`

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

✓ `int[] b = a;`

✗ `int[] c = 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:- String [] s = {"A", "B", "C"};

✓ Object() a = s;

Case(2):-

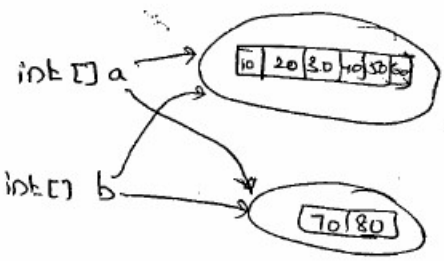
→ When ever we are assigning one array to another array only reference variables will be reassign but not underlying elements. Hence types must be matched but not sized.

Eg:- (i) int[] a = {10, 20, 30, 40, 50, 60};

int[] b = {70, 80};

✓ (1) a = b;

✓ (2) b = a;



Eg(2):

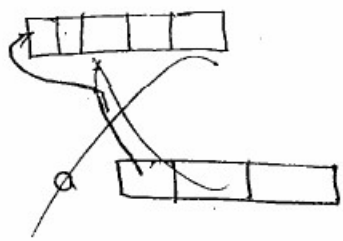
int[][] a = new int[3][2];

a[0] = new int[5];

a[1] = new int[4];

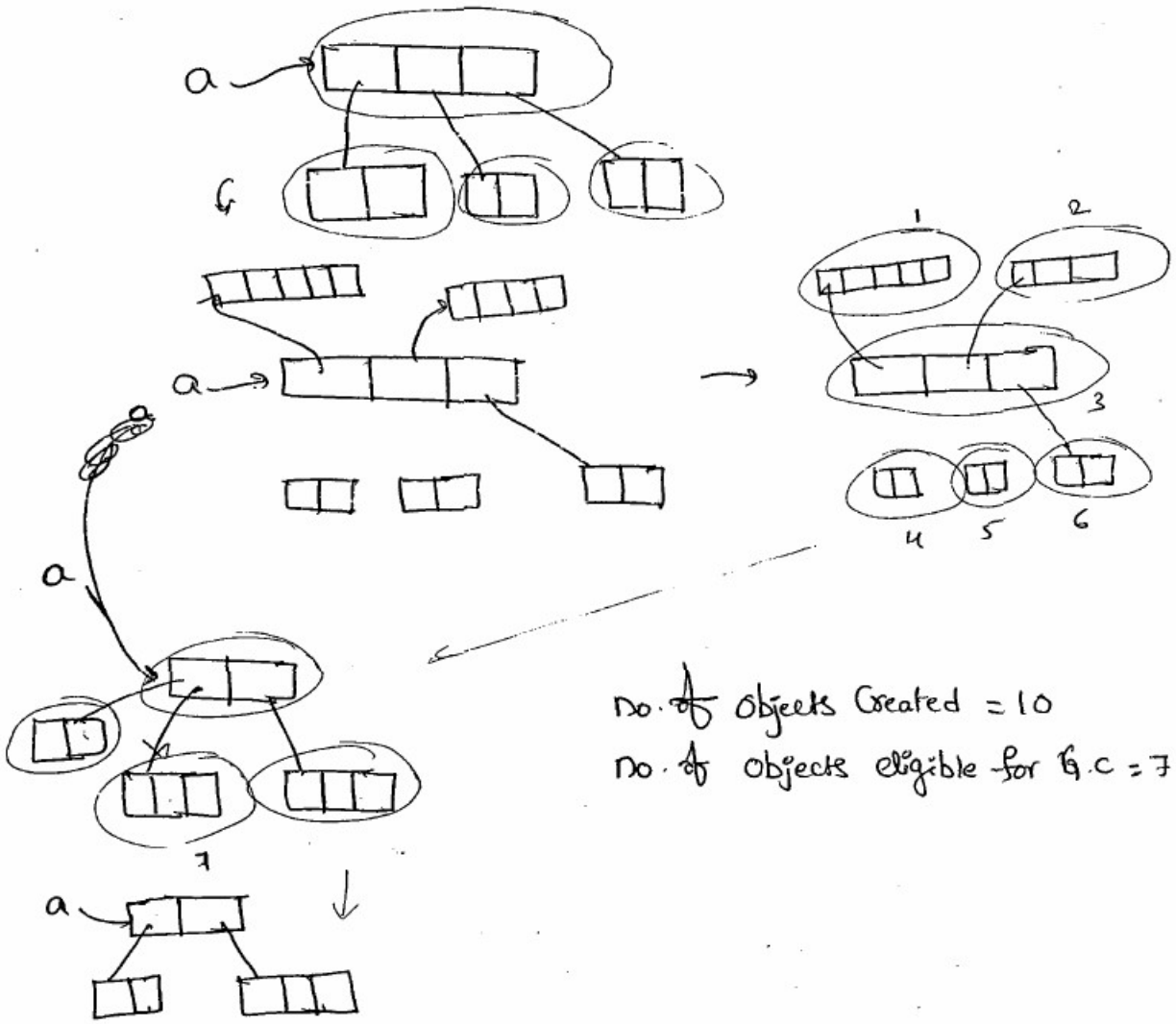
a = new int[2][3];

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:-

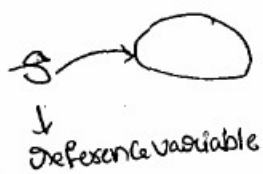
→ Can be used to represent primitive values

ex:- int x = 10;

(ii) Reference variables:-

→ Can be used to refer Objects

ex:- Student s = new Student();



→ 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 coiths in the class directly, But outside of any method or Blocks 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

Eg!.

```
Class Test
{
  int x=10;
  P.S.V.M(String[] args)
  {
   S.O.P(x); → C.E:- non-static variable x Cannot be referenced from Static context
```

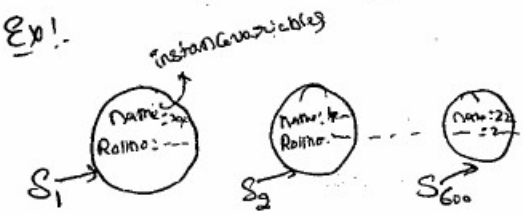
```
    Test t = new Test();
    S.o.pln(t.x); 10
  }
  public void m1()
  {
    S.o.pln(x); 10
  }
}
```

→ 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!-



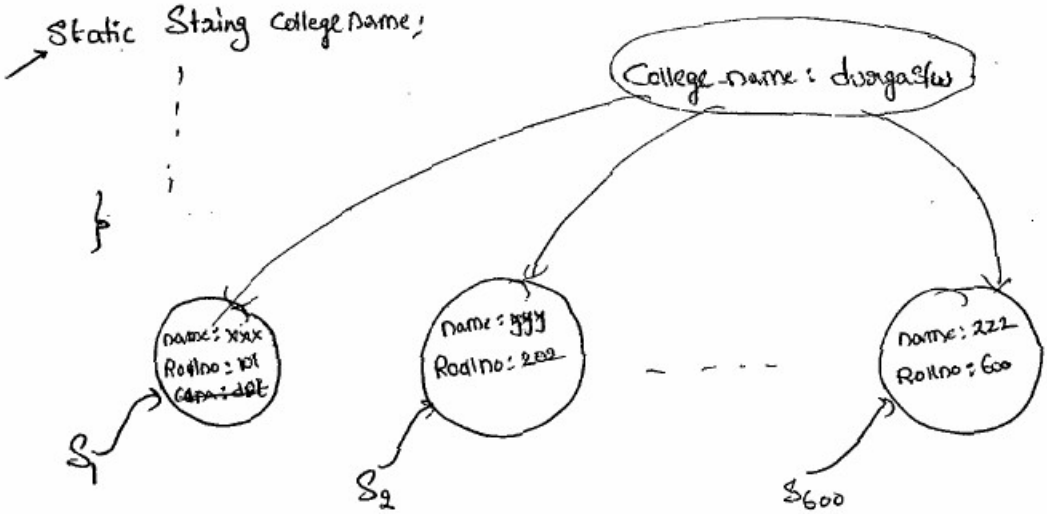
Students objects, in that name, Rollnos are instance variables, Bcz, these values are varied from object to object.

→ 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 Blocks or Constructor), with Static-modifier.

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

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

Ex:-
```
class Test
{
    static int x=10;
    p.s.v.main(String[] args)
    {
        S.o.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:-

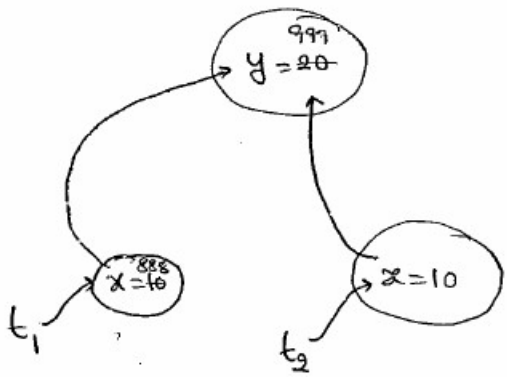
```
Class Test
{
  Static int x;
  p.s.v.m(String[] args)
  {
    S.o.pln(x); o
  }
}
```

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

Ex:

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

(t2.x → 10, t2.y → 999)



t1.x = 888
t1.y = 999

t2.x = 10
t2.y = 999

↳ If we performing any change for instance variables these changes wont be reflected for the remaining objects. because, for every object a 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 temporary requirements of the programmer 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 temporary variables.

→ Local variables will be stored inside a Stack.

→ The Local variables will be created while executing the blocks 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, Compulsory we should perform initialization Explicitly, before using that variable.

Eg:- ①

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

o/p:- Hello

```
class Test
{
  P.S.V.m(String[] args)
  {
    int x;
    S.o.pln(x);
  }
}
```

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

Eg(2) :-

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

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

Eg 3:-

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

O/P:- Java Test ↵

20

Java Test x y ↵

10

→ Note:-

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

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

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

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

Eg:-

```
Class Test
{
 p.s.v.m(String[] args)
 {
 ✗ private int x=10;
 ✗ public int x=10;
 ✗ 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.println(t1.a); null
 S.o.println(t1.a[0]); NullPointerException
}}
```

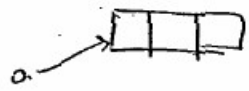
Instance level :-

int [0] 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]; So.p(a) [I@1234

So.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 (SCJP) **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 method 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.

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

Ex(1):-

```
Class Test
{
  p.s.void m1(int... i)
  {
    s.o.println("var-arg method");
  }
  p.s.v.m(String[] args)
  {
    m1();
    m1(10);
    m1(10, 20);
    m1(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 differentiate arrguements by using index.

Ex!-

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

Op!-

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

Case(1):-

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

m1(int... x) ✓

m1(int x...) X

m1(int ...x) ✓

m1(int. ..x) X

m1(int .x..) X

Case 2:.

→ We can mix var-arg parameter with normal parameters also.

Ex:- m1(int x, String... y) ✓

Case 3:.

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

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

Case 4 :.

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

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

Case 5:-

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

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

Case 6:-

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

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

Var-arg Vs Single dimensional arrays:-

Case(1):-

→ Where ever Single dimensional 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 dimensional 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 X 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 arguments

→ 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. i.e String...

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

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

(i) final

(ii) Synchronized

(iii) strictfp.

Ex:-

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

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

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

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

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

(iv) public final static void main(String[] args) ✗

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

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

Ans: No where, All cases will Compile.

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

Ex:-
```
class P
{
 public static void main(String[] args)
 {
   S.opln("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(" durgas/w");
 }
}
```

```
javac P.java
java P
o/p: I love
java C
o/p: durgas/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(" durgas/w");
 }
 p.s.v.m(int[] args)
 {
  S.o.pln(" is good");
 }
}
```

o/p:- durgas/w.

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

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

Q) Explain about S.o.pln?

A)

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

Test.name.length()
```

- 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

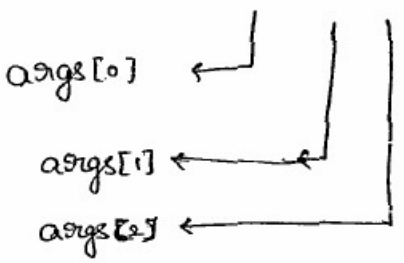
Command Line Arguements

Command line arguements :-

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

→ The main objective of Commandline 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 arrguments 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 arrguments, if the Command-Line arrguments itself Contain Space then we should enclose with in doubleCodes (")

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

Java Test "Note Book"

Ex(3):-
```
class Test
{
  p.s.v.m(String[] args)
  {
    String[] argh = {"A", "B"};
    args = argh;
    for(String S1 : args)
    {
      S.o.pln(S1);
    }
  }
}
```

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

Note: The maximum allowed no. of Commandline 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;
}
}
```

-Anonymous Standard

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

-Hitech-City

Coding Standards for Classes:-

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

Ex!- Student
Customer
String
StringBuffer,
} → Nouns

2) Coding Standards for Interfaces :-

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

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

Note :-
Throwable is a class but not interface. It acts as a root class for all Java Exceptions & Errors.

3) Coding Standards for Methods :-

→ Usually method names are either Verbs or Verb noun Combination Should Starts with LowerCase Letter & if it Contains multiple words Every inner words should starts with UpperCase Letter. (CamelCase).

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

getName()
setSalary()
} Verb + noun

(4) Coding Standards for Variables :-

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

Ex! Name
RollNo
MobileNumber
!
} → nouns

③ 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
```

④ 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". Compolsary the method should take some argument. Return type should be void.

Syntax for getter method :-

→ the method name should be prefixed with "get".
→ It should be no argument method.
→ 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 arguments 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)

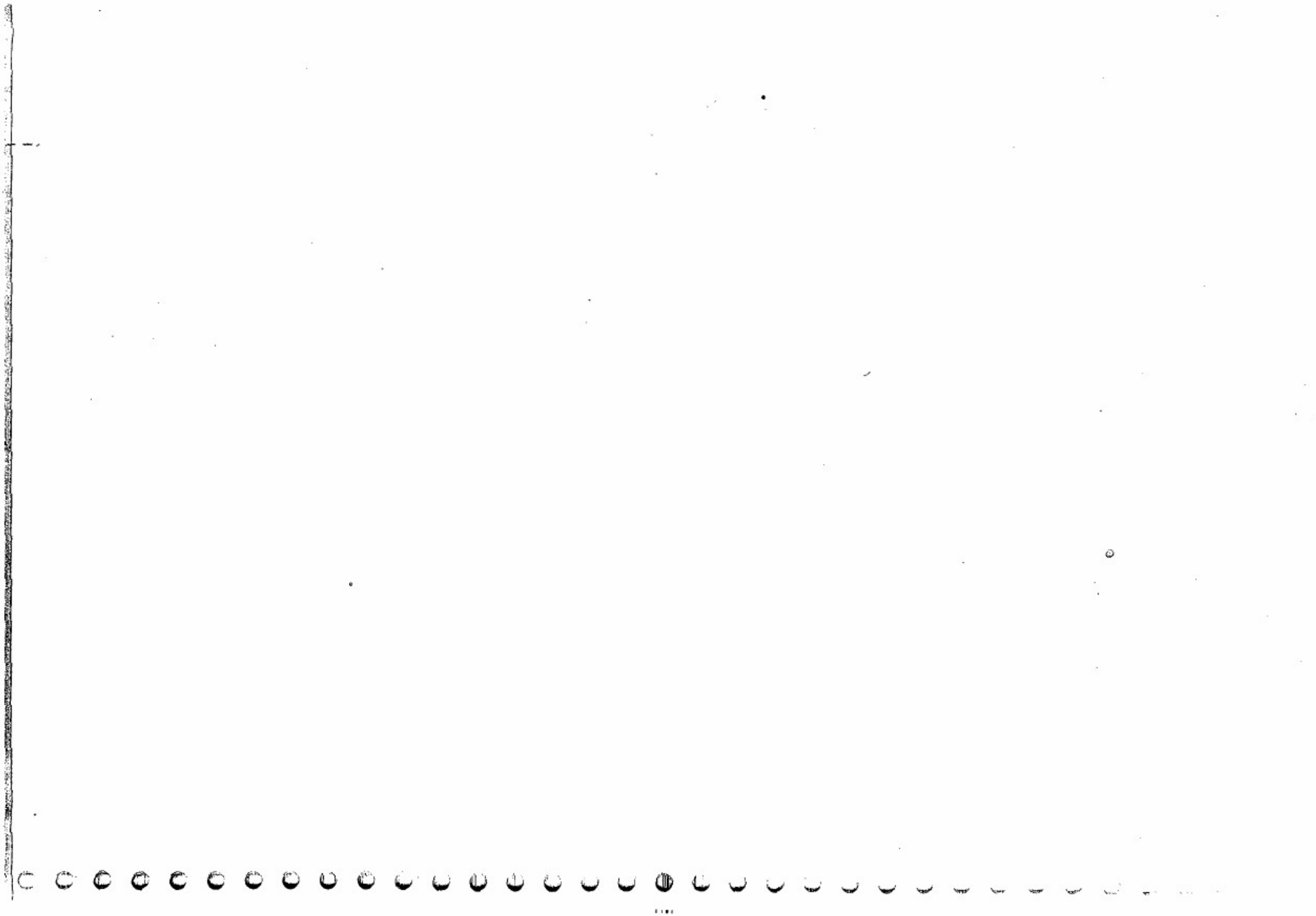
✗ ② public void unregisterMyActionListener(myActionListener l)

✗ ③ public void deleteMyActionListener(MyActionListener l)

✗ ④ public void removeMyActionListener(ActionListener l)

Note:-

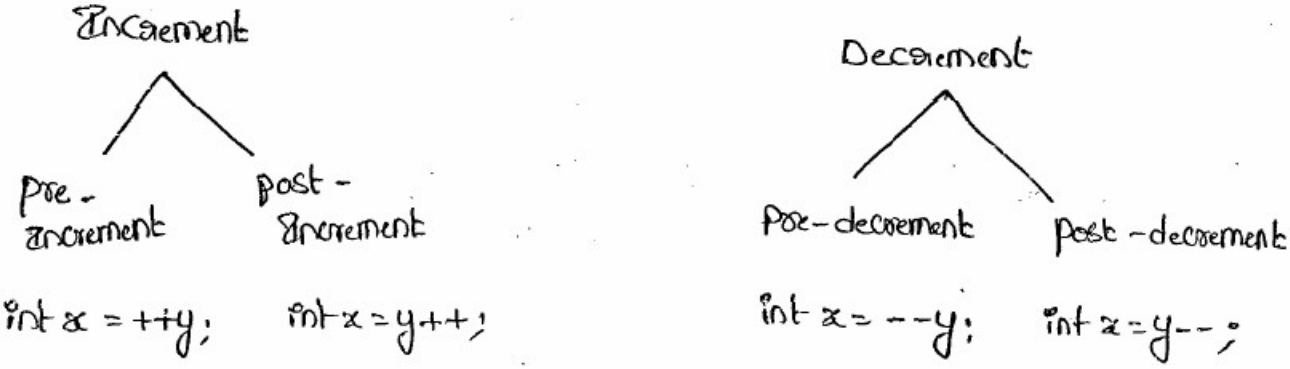
In Javabean coding standards & Listener Concept 1 compulsory.



Operators & Assignments

Kathy Sierra 1.6
book for SCJP.

Increment & Decrement Operators:



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.o.println(y);
```

C.E: unExpected type
found : value ②
required : variable ①

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

```
int x = 4;
X int y = ++(++x);   // after ++x it is constant, then
S.o.p(y);
```

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
x++;

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

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

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

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

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

③ X
```
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); 11
```

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,

> 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);
```

Arithmetic operators:-

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

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

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

byte + byte = int

byte + short = int

int + long = long

long + float = float

double + char = double

char + char = int

```
S.o.pln(10+0.0); // 10.0
S.o.pln('a'+'b'); 195
S.o.pln(100+'a'); 197.
```

Infinity:-

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

eg:-

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

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

Positive_Infinity = Infinity
Negative_Infinity = -Infinity

+ve-∞ = ∞
-ve-∞ = -∞

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

Eg:- ① S.o.pln (10/0.0) ; Infinity

② S.o.pln (-10/0.0) ; -Infinity.

* NaN :- (Not a Number)

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

Eg:- S.o.p(0/0) ; RE: 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 Constant.

→ 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.

X != NaN ⇒ True

at x=10

S.o.p(10 > float.NaN); false
S.o.p(10 < float.NaN); false
S.o.p(10 == float.NaN); false
S.o.p(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 CompileTime Error.

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

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

5. String Concatenation 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)

× 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.p(s1 == s2); false (reference)

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

Equality Operators (==, !=)

→ These are ==, !=

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

*** Eg:-

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

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

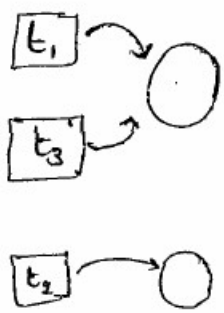
→ For the two object references r_1 and r_2 & $r_1 == r_2$ returns True iff both r_1 & r_2 are pointing to the same object.

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

Ex①: Thread t_1 = new Thread();

Thread t_2 = new Thread();

Thread t_3 = t_1;



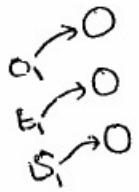
✗ S.o.p (t_1 == t_2) ; false

✓ 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].

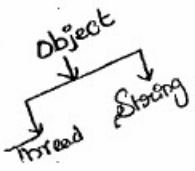
Eg:-(3):- object o_1 = new Object(); because object is Super class



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

→ So, null == null is always True.

Note:-

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

Instanceof operator (instanceof)

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

Syn:- r instanceof X

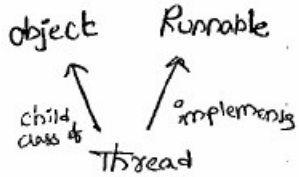
r: any reference type

X: class / interface.

instanceof
HashTree
StrictFp

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

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



✓ S.o.p(t instanceof Thread); True

✓ S.o.p(t instanceof Object); True

✓ S.o.p(t instanceof Runnable); True

↳ To use instanceof operator, compulsory there should be some relationship b/w argument type, otherwise we will get Compile-time Error saying Inconvertable type.

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

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

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

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

✓ S.o.p(o instanceof String); false

↳ For any class or interface X, null instanceof X always returning "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) arguments are True then Result is True

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

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

Ex:- 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 = 1 111 1111 ---- 1011

0 → +ve
1 → -ve

2's Complement

-ve

One's Comp

000 0000 ---- 0100

2's Comp + 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. whereas -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 e_2

10 min 1 min 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);
```

Op:-

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

(9)

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

Ans:

a) C.E

b) R.E : Arithematic Exception : / 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:-

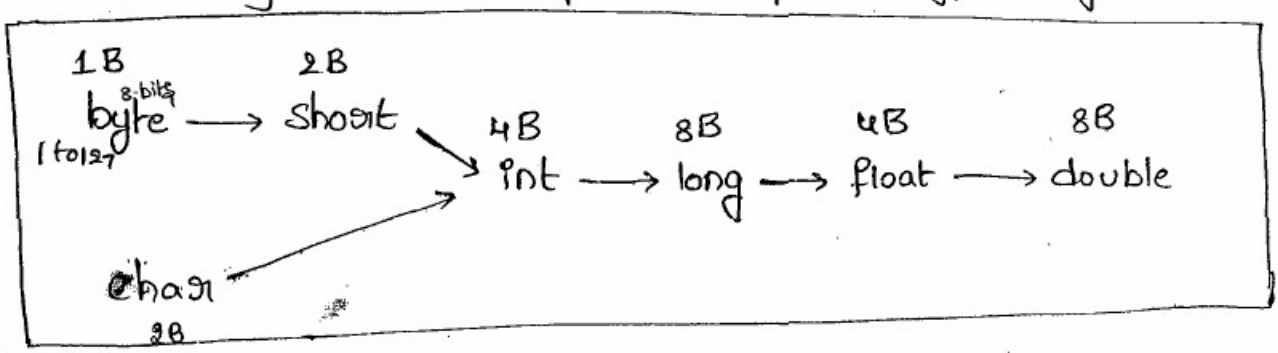
→ there are 2 types of primitive type Castings.

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 :-

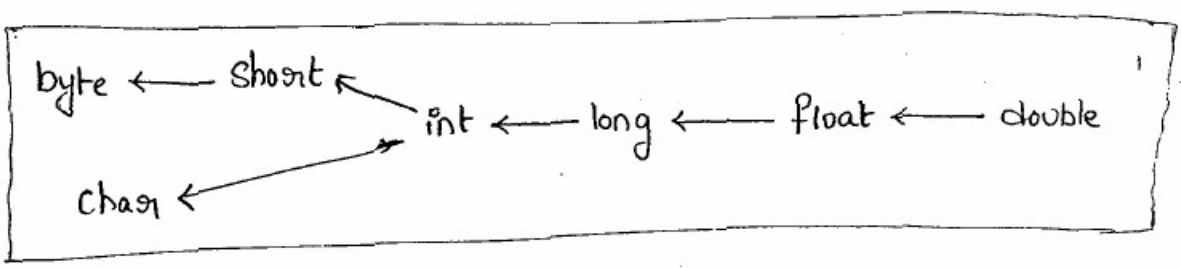
1) Programmer is responsible to perform this TypeCasting

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) x | byte b = 130

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

2) byte b = (byte) 130;

S.o.p(b); -126

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

① ✗ byte b = 130 ;

✓ byte b = (byte) 130 ;

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

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

byte b ≡ 10000010 (8 bit)

-ve ← 2's Complement

0000010 ← 1's

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

-ve 126

∴ -126

②

int i = 150 ;

Shoat S = (shoat) i ;

S.o.pln(s) ; 150

150 ≡ 0000------ 10010110 (32 bits)

Shoat S ≡ 0000 ... 10010110 ⟶ 2 Bytes = shoat = 16-bits

don't apply 2's Comp.

+ve

∴ S = 150

③ int x = 150 ;

byte b = (byte) x ;

Shoat S = (shoat) x ;

S.o.pln(b) ; -106

S.o.pln(x) ; 150

150 ≡ 0000 - - - 0 10010110

byte x = 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 ; } ✗ C.E

C.E: Can't find Symbol

Symbol : variable b

location : Class Test

int a = b = c = d = 20 ;
(Same as c & d)

Ex:- ② int b, c, d;
a = b = c = d = 20 } ✓

3. Compound assignment operator :-

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

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

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

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

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

(10)

* 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);
```

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

```
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

$1 + 2 * 3 + 4 * 5 / 6$

$1 + 6 + 4 * 5 / 6$

$1 + 6 + 20 / 6$

$1 + 6 + 3$

$7 + 3$

$= 10$

Ex(2) :-

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

1st increment
2nd place init into x

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

1st place x = 10
∴ x = 10++
↳ x = 11
but last operation is
x = 10

Ex(3) :-

(*) int x = 0;

$x = \underset{1}{++x} + \underset{1}{x++} + \underset{2}{x++} + \underset{4}{++x}$;

S.o.p(x); 8

x = 0 x x 8 4

x++ = 1
x++ = 2
3
4

Ex 4 :-

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

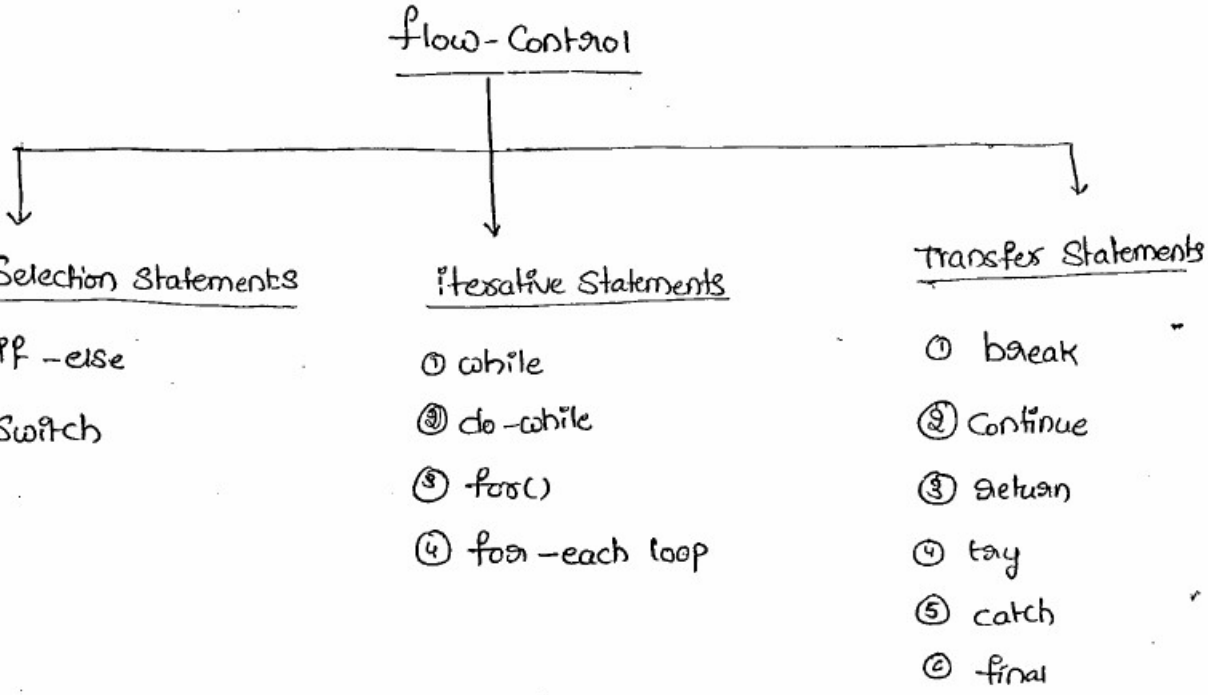
x = x + ++x + x++;
= 0 + 1 + 1
x = 2

Flow Control

16/05/2011

Flow Control :-

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



(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 Curlybraces we can take only one statement & which should not be declarative statement

Eg:-

```
if(true)
  S.o.println("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;
  Case2 :
        Action2;
    :
  default :
        default Action;
}
```

↦ Curlly braces are mandatory.

→ both Case & default are optional inside a Switch

Eg:-
```
int x=10;
Switch(x)
{
}
```
✓

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

Eg:-

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

C.E:-

Case, default or '}' expected

→ until 1.4V the allowed datatypes for switch argument are

byte
Short
int
char

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

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

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

Ex:-

```
byte b=10;
switch(b)
{
}
```
✓

```
char ch='a';
switch(ch)
{
}
```
✓

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

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

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

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

→ every case label should be within the range of switch argument type otherwise we will get Compiletime Error.

Ex:-

```
byte b=10;
switch(b)        → byte
{
case 10:
    S.o.pln("10");
case 100:
    S.o.pln("100");
case 1000:                    -128 to 127
    S.o.pln("1000");
}
```

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

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

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

Eg:-

```
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 argument & case label but case label should be constant expression

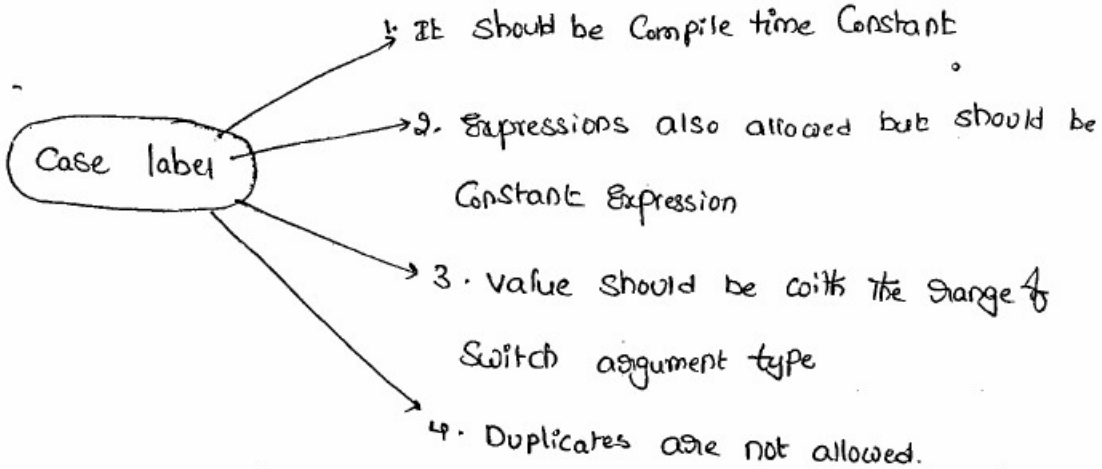
Eg:-

```
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:-



fall-through inside switch :-

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

Ex:-

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

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

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

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

o/p:-

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

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

Eg:-
```
Switch (x)
{
  Case 3:
  Case 4:
  Case 5:
        S.o.pln ("Summer");
        break;
  Case 6:
  Case 7:
  Case 8:
  Case 9:
        S.o.pln ("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.

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

x = 0 → 0
x = 1 → 1, 2
x = 2 → 2
x = 3 → def, 0

(b) Iterative Statements :-

(1) while :-

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

Ex:- ① `while (rs.next())` — `rs` : ResultSet

② `while (itr.hasNext())` — `itr` : Iterator

③ `while (e.hasMoreElements())` — `e` : enumeration

Syntax :-

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

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

Ex:-

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

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

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

Ex①

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

```
while(true);
```
✓

```
while (true)
    int x=10;
```
✗

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

Ex②:

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

C.E:- unreachable statement

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

C.E:- unreachable statement

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

o/p:- Hello
Hello
Hello
!
!

④
```
final int a=10, b=20;
while (a<b)
{
    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);
```

b → should be boolean type

; → mandatory

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

Ex:- ①

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

✓

② ; is a valid java statement

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

✓

③

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

✗

④

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

✓

⑤

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

✗ C.E:-

→ Compulsory one statement declaration take ";"

⑥

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

✓

(or)

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

o/p:- Hello
Hello
:

note:- ";" is a valid java statement

Eg:- ①

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

C.E:- unreachable Statement

②

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

o/p:- Hello
Hi

③

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

o/p:- Hello
Hello
∞

④

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

o/p:- Hello
Hi

⑤

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

C.E:- unreachable Statement

⑥

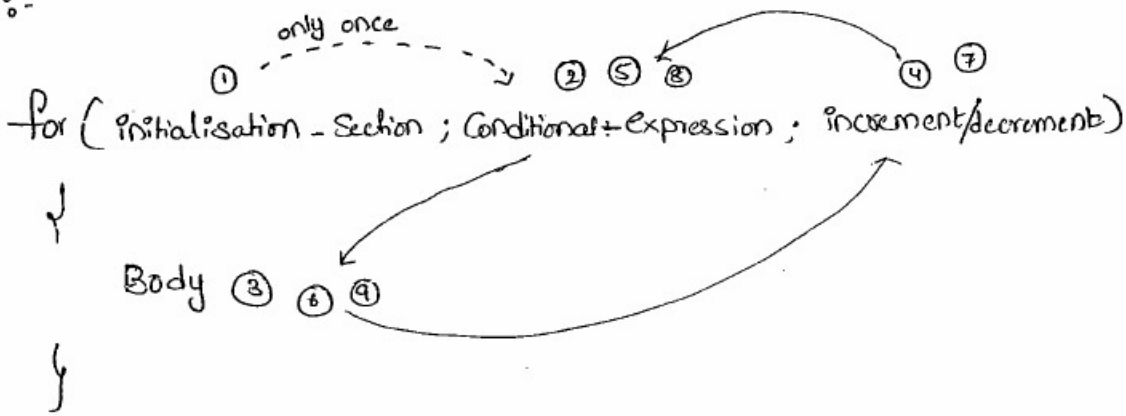
```
final int a=10, b=20;
do
{
S.o.pln("Hello");
}
while (a>b);
S.o.pln("Hi");
```

o/p:- Hello
Hi

for() :-

→ This is the most Commonly used loop

Syntax :-



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

(a) initialization-Section :-

→ This will be executed only once.

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

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

Ex:- ① int i=0, j=0; ✓

② int i=0, byte b=0; ✗

③ int i=0, int j=0; ✗

→ In the initialization Section we can take any valid java statement including S.O.P() also

Ex!- int i=0;

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

O/P!- Hello UR Sleeping

No Boss U only Sleeping
No Boss U only Sleeping
No Boss U only Sleeping

Conditional Expression:-

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

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

Increment & decrement Section :-

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

Ex!- int i=0;

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

O/P!- Hello
Hi
Hi

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

→ All 3 parts of for loop are optional

Ex!- `for(;T;);` → Statement & 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++)
```
True
```
{
  S.o.pln("Hello");
}
S.o.pln("Hi");
```
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

Eg:- 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 convient 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 retrive all values of Arrays & Collections and can't be used to retrieved a particular set of values.

(C) Transfer Statements :-

(1) break :-

→ We can use break Statement in the following cases

(i) within the Switch to stop fall through

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

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

Eg:-

```
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;
    }
  }
}
```

Ex2:-
```
l1:
for(int i=0; i<3; i++)
{
  for(int j=0; j<=3; j++)
  {
    if(i==j)
      break;
    S.o.pln(i+"-----"+j);
  }
}
```

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

break l1:-

No output

Continue:-
```
0 --- 1
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

Imp Note!:

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

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

O/P!. Hello

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

O/P!- C.E :-

unreachable statement

Declarations & Access Modifiers

Packages - 2

① Java Source file Structure (1-9)

② Class modifiers (10-14)

③ member modifiers (15-23)

④ Interfaces. (24-31)

Java Source file Structure :-

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

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

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

Save: Sri.java ✓
R.java ✓
D.java ✓

Case(1):-

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

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

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.

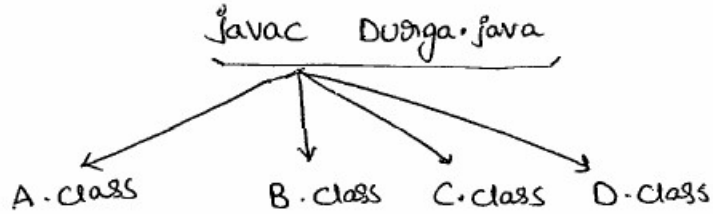
" Class A is public should be declared in a file named A.java".

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



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