# Operators & Assignments

Kathy Sierra 1.6
book for SCJP.

# Increment & Decrement Operators:.

Increment

pre-increment: int x = ++y;

post-increment: int x = y++;

Decrement

Pre-decrement: int x = --y;

post-decrement: int x = y--;

| Expression | Initial value of x | final value of x | final value of y |
|---|---|---|---|
| y = ++x; | 4 | 5 | 5 |
| y = x++; | 4 | 5 | 4 |
| y = --x; | 4 | 3 | 3 |
| y = x--; | 4 | 3 | 4 |

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

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

C.E: unExpected type

found : Value ②

required : Variable ①

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

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

after incre- it is constant then

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

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

Ex(1):- final int x = 4; ✗
x++;

Ex(2):- final int x = 4; ✗
x = 5

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

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

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

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

③
```
boolean b = true;
✗ ++b;
S.o.pln(b);
```

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

✓ ④
```
int x = 10;
x++;
S.o.pln(x); 11
```

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

① ✓

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

② ✗

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

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

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

③ ✓

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

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, typeof a, typeof b)

Max(int, byte, byte)

result is of type : int

∴ found is int but required is byte

(+, -, *, %, /)

→ whenever we are performing any arithmetic operation between two variables a & b the result type is always,

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

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

→ 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);
```

## Arithematic operators:-

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

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

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

| | |
|---|---|
| byte + byte = int | |
| byte + short = int | S.o.pln (10 + 0.0); // 10.0 |
| int + long = long | S.o.pln ('a' + 'b') ; 195 |
| long + float = float | S.o.pln (100 + 'a') ; 197 . |
| double + char = double | |
| char + char = int | |

## Infinity:-

→ In the Case of integral arithematic (int, short, long, byte), there is no way to represent infinity. Hence, if the infinity is the result we will always get ArithematicException. (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 ~~fleat~~ floating point Arithematic we wont get any Arithematic Exception.

Eg:- ①. S.o.pln (10/0.0) ; Infinity

②. S.o.pln (-10/0.0) ; -Infinity.

* NaN :- (Not a Number)

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

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

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

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

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

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

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

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

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

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

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

x > NaN<br>
x >= NaN<br>
x < NaN<br>
x <= NaN<br>
x == NaN } false

$\boxed{at\ x=10}$

```
S.o.p (10 > float.NaN);          false
S.o.p (10 < float.NaN) ;         false
S.o.p (10 == float.NaN);         flase
S.o.p (10 != Float.NaN);         true.
S.o.p (float.NaN == float.NaN);  false
S.o.p(Float.NaN != float.NaN);   True.
```

Conclusion about A.E (ArithmeticException) :-

→ It is RuntimeException but not Compile time Error.

→ possible only in Integral Arithematic but not floating point Arithmatic

(int, byte, short, char)   (float, double)

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

## 3. String Concatination Operator (+)

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

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

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

String d = "shanth";

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

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

✓ $\underset{String}{c} = \underbrace{a+c}_{total\ String}$;

✓ $\underset{int}{b} = \underbrace{a+b}_{int}$;

× 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 ✓ 5) true <= true

2) 'a' < 'b' True ✓ 6) true < false

3) 10 >=10.0 True ✓ CE:- operator <= can't be applied to boolean, boolean

4) 'a' < 125 True ✓

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.

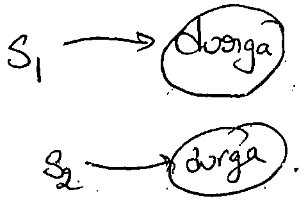
3) Nesting of Relational operators we are not allowed to apply.

Eg:- ✓ S.o.p (10 < 20);

✗ S.o.p $(\underbrace{10<20}_{\text{boolean}} <30)$

CE!- operator < can't be applied to boolean.

Eg:- String s₁ = new String("durga");
String s₂ = new String("durga");



S.o.pln( S₁==S₂); false (reference)

S.o.pln(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

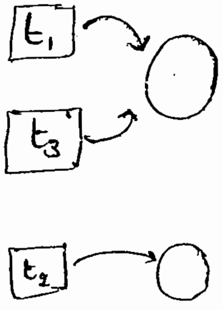
Ex①: Thread $t_1$ = new Thread();

Thread $t_2$ = new Thread();

Thread $t_3$ = $t_1$;

× S.o.p ($t_1$ = = $t_2$) ; False

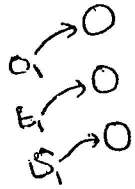
✓ S.o.p ($t_1$ = = $t_3$) ; True



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

[either parent to child (or) child to parent (or) Same type] otherwise we will get CE: InComparable type ]

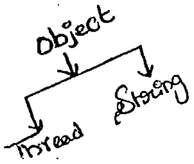
Eg:-(2):- object o1 = new Object(); because object is Super class



Thread t1 = new Thread();

String s1 = new String("shanth");

S.o.p(t1 == s1); CE:- InCompatiable types java.lang Thread & java.lang. String



S.o.p(t1 == o1); F

S.o.p(s1 == o1); F

→ for any object reference ~~r~~ x, 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 ment for reference Comparision where as .equals() method ment for Content Comparision.

## Instanceof operator (instanceof) ✓

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

Syn:- r instanceof X

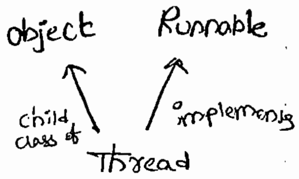
r → any reference type

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, Compulsary 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 ~~of~~ interface ~~of~~ X, null instanceOf X always returns "false".

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

eg:-
```
Iterator itr = l.iterator();
while (itr.hasnext())
{
Object o = itr.next();
if (o instanceOf Student)
{ Apply Student related function }
elseif(o instanceof Cu)
{ Apply Customer related
}
```

## Bit-wise Operators :-

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

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

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

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

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

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

Ex(1):- S.o.pln(4&5); 4 → $\begin{array}{r}100\\101\\\hline 100\\\hline\end{array} = 4$

S.o.pln(4|5); 5 → $\begin{array}{r}100\\101\\\hline 101\\\hline\end{array} = 5$

S.o.pln(4^5); 1 → $\begin{array}{r}100\\101\\\hline 001\\\hline\end{array} = 1$

→ 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

(1) 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 = $\boxed{1}$111 1111 ---- 1011

2's Complement

-ve

0 → +ve

1 → -ve

One's Comp

000 0000 ---- 0100

0 1

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

2's Comp 000 -------- 0101

-ve 5

$\boxed{\therefore -5}$

Note:

→ the most Significant bit represents Sign bit. 0 means +ve no, 1 means -ve no.

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

## Boolean Complement Operator (!) :-

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

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

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

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

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

Summary:-

& <br>
| <br>
^ ⇒ 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 ( ~~~~~  &  ~~~~ )
     e1     ↓    e2
   10min   1min  10 min
      \_____+_____/
          21 min
{
}
else
{
}
```

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

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

Ex:-

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

o/p:-

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

②

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

Ans:

a) C.E

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

c) Hello

d) Hi

Note:

if we Replace && with &

then Result is ⓑ, that is R.E.

a=97
A=65

## TypeCast Operators:-

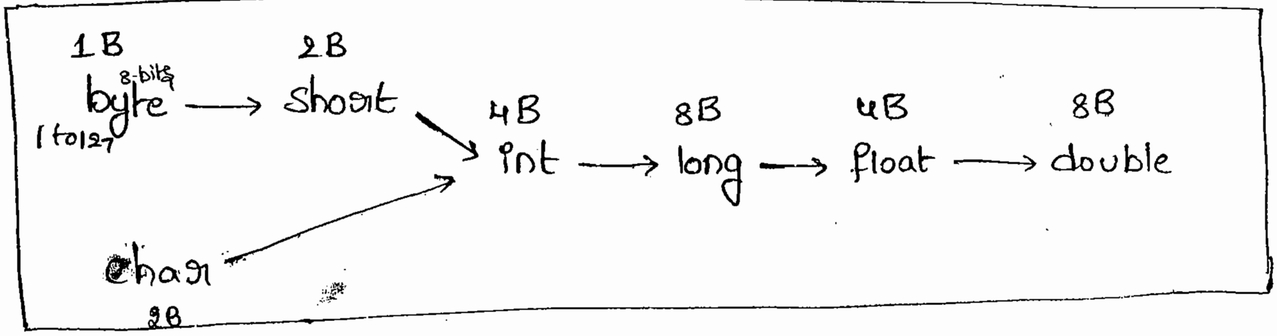
→ There are 2 types of primitive type Castings.

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; [ Complier 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 :-

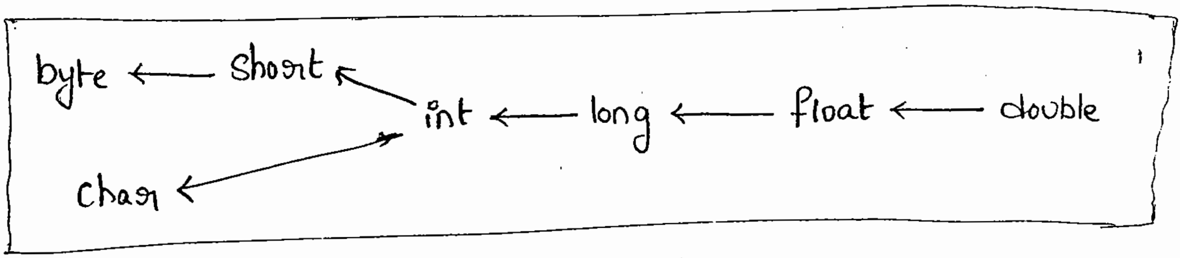
1) Programmer is responsible to perform this TypeCasting

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 TypeCasting.

→ 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 X byte b = 130 ;

✓ byte b = (byte) 130 ;

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

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

byte b ≡ 10000010 (8 bit)

-ve

2's Complement

0000010

1111110

1111101
+ 1
1111110

= 1×2⁶ + 1×2⁵ + 1×2⁴ + 1×2³ + 1×2² + 1×2¹ + 0×2⁰

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

~~+ve~~ 126

∴ -126

② int i = 150 ;

short s = (short) i ;

S.o.pln (s) ≠ 150

150 ≡ 0000 - - - - - 010010110 32 bits

short s ≡ 0000 - - - 0010010110 ⟶ 2 Bytes = short = 16-bits

+ve

~~do~~ don't apply 2's Comp.

∴ S = 150

③ int x = 150 ;

byte b = (byte) x ;

short s = (short) x ;

S.o.pln (b) ; -106

S.o.pln (x) ; 150

150 ≡ 0000 - - - 010010110

byte ~~x~~ = 10010110

-ve

~~x~~ 2's Com

1101001
+ 1
1101010

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;
```

12.48

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

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

C.E: Can't find Symbol

Symbol : variable b

location : Class Test

```
int a = b = c = d = 20 ;
        ^
```
(Same c. & d)

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

3. Compound assignment operator :-

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

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

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

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

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

11

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

Ex ①

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

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

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

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

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

Ex②:-

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

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

S.o.pln(a+"----"+b+"----"+c+"-----"+d);
      620        600        30        10
```

## Conditional Operator (?:)

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

a+b → binary operator

++a → unary "

(a+b)? a:b; → ternary.

Ex:-

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

a>b is T then 40

a>b is F then 50

→ nesting of Conditional operator is possible.

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

```
int x = (a>50) ? 777 : ((b>100) ? 888 : 999));
           F               F
S.o.pln(x) ; 999
```

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

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

✓ a<12 T

✗ a<b ✗ C.E

don't compare these variables

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

C.E:- PLP

found : int

required : byte.

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

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

## New Operator :-

→ We Can use this Operator for Creation of objects.

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

## [] operator :-

→ we Can use these Operator for declaring & Creating arrays.

## Operator precidence :-

1. Unary operators :-

[ ] , x++ , x--

++x , --x , ~ , !

new , <type> (used to type cast)

2. Arithematic Operators :-

* , / , %

+ , -

3. shift operator :-

>>> , >> , <<

4. Comparision operator :-

< , <= , > , >= , instanceOf

5. Equality operator :-

== , !=

6. Bitwise operators :-

&

^

|

7. Short - Circuit operators :-

&&

||

8. Conditional operators :-

?:

9. Assignment operators :-

= , += , -= , . . . . .

## Evalution order of operands :-

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

Ex:-

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

o/p:-

10

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

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

$$1 + 6 + 20/6$$

$$1 + 6 + 3$$

$$7 + 3$$

$$= 10$$

Ex(2):-

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

1st increment

2nd place 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; $(1+2)^3$

$$x = \underbrace{\overset{\downarrow}{++x}}_{1} + \underbrace{\overset{①}{x++}}_{1} + \underbrace{\overset{③}{x++}}_{2} + \underbrace{++x}_{4};$$

S.o.p(x); 8

$x = \cancel{0}\cancel{1}\cancel{2}\cancel{3}4$

x++ = 1

x++ = 2

3

4

Ex 4:-

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

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