Date / /
Page

Expression	Initial value of x	value of y	Final value of x
y = ++x	10	11	11
y = x++	10	10	11
y = --x	10	9	9
y = x--	10	10	9

case1

→ We can apply Inc/Dec op only for Variable Not for constant values we will get C.E.

```
int x=10;
int y=++x;
s.o.p(y);
```
(11)

```
int x=10;
int y=++10;
s.o.p(y);
```

C.E.
Unexpected Type
found: value
Req: Variable

case2

→ Nesting (X)

```
int x=10;
int y=++(++x);
s.o.p(y);
```
(X)

→ Bcz After Applying Inc/Dec op (++x) becomes value & we can't Apply Inc/Dec op to value

Operators & Assignments

Page

1. Increment & Decrement Operators [++ --]
2. Arithmetic Operators [+, -, *, /, %]
3. String concatenation operator [+]
4. Relational Operators [<, <=, >, >=]
5. equality Operators [==, !=]
6. instanceof operator [instanceof]
7. bitwise operators [&, |, ^] [~]
8. short circuit operators [&&, ||]
9. type cast operator [Implicit, Explicit]
10. Assignment operator [simple, chained, compound]
11. conditional operator [a?b:c]
12. new operator [new]
13. [] operator []
14. operator precedence
15. Evaluation order of operands
16. new (vs) newInstance()
17. instanceof (vs) isInstance()
18. ClassNotFoundException (vs) NoClassDefFoundError

(*) Increment & Decrement Operator

- pre-Increment: y = ++x;
- post Increment: y = x++
- pre-Decrement: y = --x;
- post Decrement: y = x--;

Date / /
Page

~> Ex byte a = 10;
byte b = 20;
byte c = a + b; ↔ C.E. PLP F: int, R: byte
s.o.p(c);

a op b	where op = +, -, *, /, %

Resulting type = max(int, type of a, type of b)

Here, a + b

Resulting type = max(int, byte, byte)
= int

So int can't be assign to byte
(type casting needed)

∴ byte c = (byte)(a+b); ✓
Here, c = 30 ✓

~> Now, byte b = 10;
b = b+1;
s.o.p(b); ✗ → byte literal

case 3
→ Can't Apply to final variable

final int x = 10;
x++;
s.o.p(x); ✗ → C.E. cannot Assign a value to final variable x

case 4
→ Can Apply to All primitive data type except boolean.

int x = 10; x++ s.o.p(x); 11 ✓	char ch = 'a'; ch++; s.o.p(ch); b ✓	double d = 10.5; d++; s.o.p(d); 11.5 ✓	boolean b = true; b++; s.o.p(b); ✗

(C.E. operator ++ cannot be applied to boolean)

case 5
→ b = b+1 & b++ internally not same.

★

byte b = 10; b = b+1; s.o.p(b); ✗ (C.E.)	byte b = 10; b++; s.o.p(b); ✓ 11

Scanned with CamScanner

Date / /
Page

byte → short → int → long → float → double
char → int

min → max

byte + byte = int bcz max(int, byte, byte)
byte + short = int
short + short = int
byte + long = long
long + double = double
float + long = float

s.o.p('a' + 'b'); ⟹ 97 + 98 = 195
char + char = int

s.o.p('a' + 0.87); ⟹ 97 + 0.87 = 97.87
char + double = double

★ case2 ⟹ / by zero in Integral & Floating point Arithmetic

s.o.p(10/0);
s.o.p(10/0.0);
s.o.p(0/0);
s.o.p(0/0.0);
→ ?

b = b + 1;

resulting type = max(int, byte, int)
= int

∴ b = (byte)(b+1);

Now b ⇒ 11

⟹ so, here (b++) operator automatically type cast (if needed).

b++ ⟹ (type of b)(b+1)

Conclusion : In Increment / Decr. operator internal type casting will be done automatically.

⊛ Arithmetic Operators (+, -, *, /, %)

case1 ⟹ Result type, when we apply arithmetic operator to any two variable,

= max(int, type of var_1, type of var_2)

(iii) S.O.P (0/0);

Here,

In, Normal Mathematics

0/0 ⇒ Undefined

R.E ArithmeticException / by zero.

Bcz, Integral Arithmetics don't have any way to represent Undefined values.

(iv) S.O.P (0/0.0); NaN

S.O.P (-0.0/0); NaN

Floating-point Arithmetic Represent Undefined Number as NaN

(Not a Number)

Float & Double class have constant named NaN.

Case3

⟶ Behavior of NaN

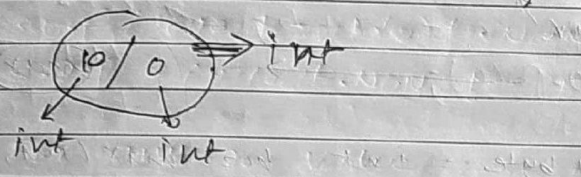
S.O.P (10 < Float.NaN); false

S.O.P (10 <= Float.NaN); false

S.O.P (10 > Float.NaN); false

S.O.P (10 >= Float.NaN); false

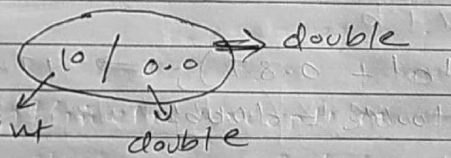
(i) S.O.P (10/0); Runtime Exception (ArithmeticException / by zero)



Integer Arithmetic (byte, short, int, long)

There is no way to represent ∞, so we will get R.E.

(ii) S.O.P (10/0.0); Infinity



S.O.P (-10/0.0); -Infinity

Floating-point Arithmetic (Float, Double) has a way to Represent ∞ by Infinity & -Infinity

So, we don't get R.E.

Bcz, Float & Double classes have constants named POSITIVE_INFINITY & NEGETIVE_INFINITY

⊛ String Concatenation Operator

⟶ Only overloaded operator in java

+ → Act as Arithmetic Addition op.
+ → Act as String concatenation op.

```
String a = "durga";
int b=10, c=20, d=30;
```

- `s.o.p(a+b+c+d);` durga102030
  `s.o.p(b+c+d+a);` 60durga
  `s.o.p(b+c+a+d);` 30durga30
  `s.o.p(b+a+c+d);` 10durga2030

- if one operand is string type then + Act as concatenation op.

- `a = b+c+d;` ✗ C.E Incompatible
  `a = a+b+c;` ✓
  `b = a+c+d;` ✗ C.E Incompatible
  `b = b+c+d;` ✓

⊛ Relational Operator (<, <=, >, >=)

⟶ We can apply relation op. to any primitive type except boolean

`s.o.p(10<20);` true
`s.o.p('a'<10);` false
`s.o.p(10.5<20.5);` true
`s.o.p(true<false);` C.E

`s.o.p(10 == Float.NaN);` false
`s.o.p(Float.NaN == Float.NaN);` false
`s.o.p(10 != Float.NaN);` true
`s.o.p(Float.NaN != Float.NaN);` true

↘ For any x value

x < NaN
x <= NaN
x > NaN
x >= NaN
x == NaN

⇓ false

x != NaN

⇓ True

Catch

⟶ ArithmeticException

① R.E Not C.E

ArithmeticException

② Only occur for Integer Arithmetic
Not occur for Floating-point Arith.

③ Only operators which cause A.E are / & %

Date / / Page

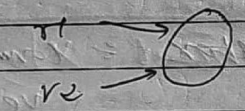
S.o.p (10==20); false
S.o.p ('a'==16); false
S.o.p ('a'==97.0); true
★ S.o.p (false==false); true

→ We can apply Equality Op. to Object types also.

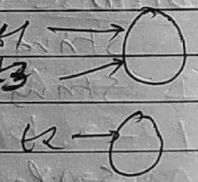
For object references:

r1 == r2 ⇒ true

if and only if both referring to same object.



Thread t1 = new Thread();
Thread t2 = new Thread();
Thread t3 = t1;
S.o.p(t1==t2); false
S.o.p(t1==t3); true



→ For comparing two object references they should have some relation with each other
(same type) (Parent-child type)

S.o.p('a'<97.6); true
S.o.p('a'>'A'); true
S.o.p(true>false); → C.E Operator > cannot be applied to boolean, boolean

→ We can't apply Relational op. to Object types

S.o.p("durgasoft">"durga"); → C.E Operator > cannot be applied to j.l.String, java.l.String

→ Nesting of Relational op. Not possible

S.o.p(10<20<30); (X)
true
C.E Operator < cannot be applied to boolean, int

⊛ Equality Operators (==, !=)

→ We can apply Equality op to all primitives including boolean.

instanceof Operator

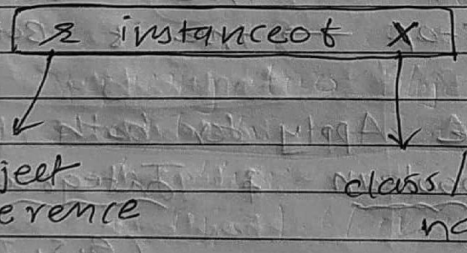
→ Here, o is (not) capital (Not camelcase) (All small)

→ We can use instanceof operator to check given object is of particular type or not.

l → [0] [1] [2] [3]

```
Object o = l.get(0);
if (o instanceof Student)
{
    Student s = (Student) o;
    // perform student-related thing
}
else if (o instanceof Customer)
{
    Customer c = (Customer) c;
    // perform customer-related things
}
```

→ Syntax:



★ (x and X should have same Relation)

```
Thread t1 = new Thread();
Object o1 = new Object();
String s1 = new String("durga");

s.o.p(t1==o1); false      t1 → ○
s.o.p(o1==s1); false      o1 → ○
s.o.p(t1==s1); ✗          s1 → "durga"
```

C.E: Incomparable Types
j.l.String and j.l.Thread

→ Difference b/w == operator & equals() method

== Operator : comparing object references

equals() method : comparing object content.

```
String s1 = new String("d");
String s2 = new String("d");

s.o.p(s1==s2); false
s.o.p(s1.equals(s2)); true
```

→ `s.o.p(null==null); true`

null with anything ⇒ false.

→ Bitwise for boolean :-

S.O.P (true & false); false
S.O.P (true | false); true
S.O.P (true ^ false); true

→ Bitwise for Integral Types :-

S.O.P (4 & 5); (4)
S.O.P (4 | 5); (5)
S.O.P (4 ^ 5); (1)

4 → (int) → 0000 ... 100 — 32 bit
5 → (int) → 0000 ... 101 — 32 bit

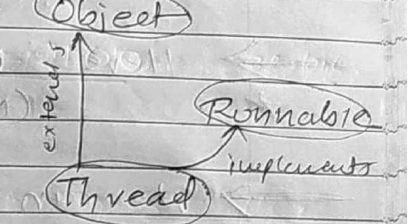
AND (&) 0000 ... 100 → (4)
OR (|) 0000 ... 101 → (5)
X-OR (^) 0000 ... 001 → (1)

→ Bitwise complement Operator (~)

S.O.P (~ true); C.E: operator ~ cannot be applied to boolean

So, (~) can only be Applied to Integral Type, Not for boolean

Thread t = new Thread();



S.O.P (t instanceof Object);
S.O.P (t instanceof Thread);
S.O.P (t instanceof Runnable);

O/P → true, true, true

S.O.P (t instanceof String);

↳ C.E: Inconvertible Types
F: j.l.Thread
R: j.l.String

S.O.P (null instanceof X); ⟹ false

where X = Any class/interface

Bitwise Operators (&, |, ^)

→ & → AND → true if Both true
| → OR → true if one true
^ → X-OR → true if Both different

~ Can be Apply for both boolean & Integral Types (B, S, I, L)

⊛ Short-Circuit Operator (&&, ||)

→ These are same functional as Bitwise Operator except following difference:

&, |	&&, ||
- Both Arguments evaluated Always	- Second Argument evaluation is based on 1st argument
- Relative Performance Low	- Relative performance High
- Applicable for both • Integral & • Boolean	- Applicable only for Boolean.

x && y : y only evaluated if x is true

x || y : y only evaluated if x is false

⊛ Type-Cast Operator

S.O.P(~4); → -5

4 → int → 0000...0100 (32-bit)

sign bit, value

~4 = 1111...1011

Do 2's comp

000...0100 → 1's comp

+ 1

1000...0101

-5

→ Boolean complement Operator (!)

S.O.P(!4) → C.E. operator ! cannot be applied to int

S.O.P(!false); true

	Applicable	Not
&, |, ^, ~	boolean, Integral	- boolean, Integral
!	boolean	

→ Explicit Type Casting

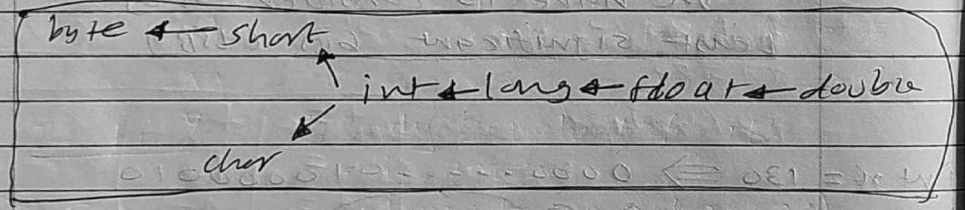
1) Programmer is Responsible to perform Explicit type casting.

2) Whenever we assigning Bigger data type value to smaller data type variable, this happen

3) This is known as "Narrowing" or "DownCasting"

4) In this, there might be Loss of Information.

→ Various conversions where Explicit type casting required



→ E.g. int x = 10;

byte b = x; (X) → C.E: PLP
F: int, R: byte

(Or int → 4 byte, byte → 1 byte)

byte b = (byte) x; (✓)

S.O.P(b); 10

(Here, No Loss bcz 10 is in range of byte)

→ Implicit Type Casting

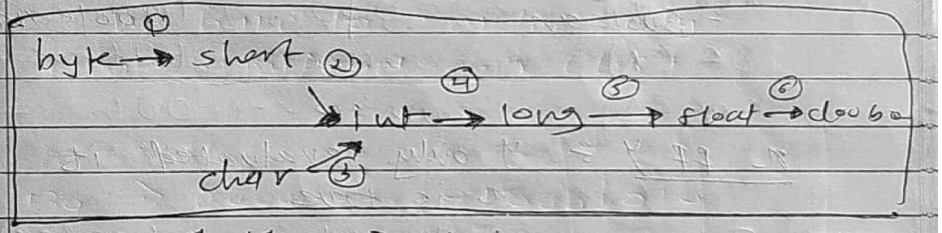
1) Compiler is Responsible to perform implicit type casting

2) Whenever we assigning smaller data type value (to) bigger data type variable, this happen.

3) This is known as "Widening" or "Upcasting"

4) In this, No Loss of Information

→ Various conversions where Implicit type casting possible



→ Eg.

int x = 'a'; (97) compiler converts (char) to (int) Automatically

S.O.P(x);

double d = 10; (10.0) compiler converts (int) to (double) Automatically by Implicit Type Casting

S.O.P(d);

int x = 10

byte b = (byte) x;

S.O.P(b); ⇒ 10

int x = 10; ⇒ 0000 ... 00001010

byte b = (byte)x; ⇒ 0 0000 1010

2 | 10	
2 | 5 0	sign bit → 0 → +ve
2 | 2 1	original value ∴ 10
1 0	

⇒ If we assign floating-point values to Integral values, digits after decimal point are gone.

```
double d = 120.456;
int x = (int)d;
S.O.P(x);   (120)

byte b = (byte)d;
S.O.P(b);   (-120)
```

⊛ Assignment Operator

⇒ Three types of Assignment operator

① simple ② Chained ③ Compound

```
int x=10;        int a,b,c;        int x;
                 a=b=c=20;         x+=10;
```

Date / / Page

```
int x = 130;
byte b = x;   (X)   C.E: PLP
                    F: int R; byte
byte b = (byte)x;
S.O.P(b);   (-126)
```

Here, loss of info is there bcz 130 is out of range of byte, but still we are enforcing the casting

⇒ When we assign bigger data type value to smaller data type variable then by explicit typecasting the Most significant bits (MSB) will be lost

We have to consider only Least significant bits (LSB)

int x = 130 ⇒ 0000 ... 10000010

byte b = (byte)x ⇒ 1 0000010

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

sign-bit → 1 ⇒ -ve

As sign-bit is 1 ⇒ -ve

2's complement form is there

So do 2's complement again to get original value

1 1 1 1 1 1 0

\+ 1

sign bit (1) 1 1 1 1 1 1 0 ⇒ (-126)

ch = ch+1

Date / / Page

→ In case of compound Assignment operator,

Internal Type casting will be done Automatically

byte b = 10;
b = b+1;
s.o.p(b);
✗
	byte b = 10;
	b++;
	s.o.p(b);
	✓
		byte b = 10;
		b += 1;
		s.o.p(b);
		✓

max(int, byte, int); = int	b = (byte)(b+1);	b = (byte)(b+1);

byte b = 127;
b += 3;
s.o.p(b); ⇒ -126 ✓

b = (byte)(b+3);

→ Eg.

int a,b,c,d;
a = b = c = d = 20;
a += b -= c *= d /= 2;
s.o.p(a + "--" + b + "--" + c + "--" + d);

-160	-120	200	10

Right to left Assignment (Always)

→ We can't perform chained Assignment with declaration

int a,b,c; ✓ a = b = c = 20;	int a = b = c = 20; ✗
Assigning flow ←	C.E: cannot find symbol s: b b = class Test

Here, we only declare a & we are using b,c for assigning value to a

int b,c;
int a = b = c = 20; ✓

→ Assignment compound operator (+=, ...)

Total CAO in JAVA

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

(11)

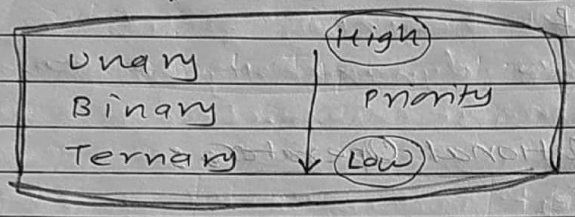
a >>> 2

0 1 0 1 0 1 1 0

sign bit ⇒ >>

always fill with 0 ⇒ >>>

Java Operator Precedence



→ Assignment Op have least priority.

→ List of precedence from Higher to Lower, (Here same line = same priority)

① Unary Operators:
[ ], x++, x--
++x, --x, ~, !
new, <type-cast>

② Arithmetic Operators:
*, /, %
+, -

③ Shift Operators:
>>, >>>, <<

④ Comparison Operators:
<, <=, >, >=, instanceof

⑤ Equality Operators:
==, !=

⑥ Bitwise Operator:
&
^
|

High → Low

Conditional Operator (? :)

- only possible ternary operator in java

- ++a; a++ ⇒ unary
  a+b ⇒ Binary
  (a) ? b : c ⇒ Ternary

- int x = (10 < 20) ? 30 : 40;
  sop(x); (30)

- int x = (10 > 20) ? 30 : ((40 > 50) ? 60 : 70)
  sop(x); (70)

  nesting ✓

new Operator

- To create new object
  Test t = new Test();
  ↳ constructor call (To perform initialization)

[No, delete keyword due to GC]

[] Operator

- To create Array / To Declare Array
  int[] x = new int[10];

```
p s int m(int i)
{
    S.O.P(i);
    return i;
}
}
```

O/P :- 1 2 3 4 5 6 32

```
1 + 2 * 3 / 4 + 5 * 6
1 + 6/4 + 5 * 6
1 + 1 + 5 * 6
1 + 1 + 30
2 + 30
(32)
```

★ Interview

(*) Differences — Functionally Equivalent

→ new vs newInstance()

Diff 1)

→ we can use (new) operator to create an Object, if we know class name at begining.

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

→ (newInstance()) is a method of class Class, we can use newInstance() method to create

(7) Short Circuit Operators :

&&

||

(8) conditional Operator (*) :

?:

(9) Assignment Operators :

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

For same priority (left) → (Right)

(*) Evaluation order of Operands

→ In java, we have only

✓ Operator Precedence But Not

✗ Operand precedence.

Before applying operators, All operands will be evaluated from (Left → Right)

→ Eg.

```
class Test
{
    p s v main(String[] args)
    {
        S.O.P(m(1) + m(2)
              * m(3) / m(4)
              + m(5) * m(6));
    }
```

→ Class.forName(args[0]).newInstance();

Returns object of class Class.

Diff-2

→ In the case of new operator based on our requirement we can invoke any constructor

★
```
Test t = new Test();
Test t1 = new Test(10);
Test t2 = new Test("durga");
```

But

newInstance() method internally calls no-arg constructor, so compulsory corresponding class should contain no-arg constructor, otherwise we will get Runtime Exception. called

R.E InstantiationException.

Diff-3

→ While using new operator, at Runtime if the corresponding .class file is not available then we will get Runtime Exception.

R.E. NoClassDefFoundError: Test

object if we don't know class name at begining & it is available dynamically at Run-Time.

```
Class Test
{
    p s v main(String[] args) throws Exception
    {
        args[0] obj = new args[0](); (X)

        Object o =

            Class.forName(args[0]).newInstance();

            S.O.p("object created for: "
                + o.getClass().getName());
    }
}
```

java Test student ↵
Object created for student

java Test Customer ↵
Object created for Customer

java Test j.l.String ↵
Object created for j.l.String

class Class is available in java.lang package

Date / / Page

Web-container creates an object of particular servlet at Runtime.

/test1 → [ ] — gets the Name of servlet (class) → create object of servlet at Run-time

url

(Hit-Url)

webxml

→ so, servlet also has to have no-arg constructor.

2. ClassNotFoundException (VS) NoClassDefFoundError

ClassNotFoundException	NoClassDefFoundError
For Dynamically provided class-name, At Runtime, if corresponding .class file is not Avail. then we will get R.E.	For Hard-coded class-name, At Run-time if corresponding .class file is Not Avail. then we will get R.E.
ClassNotFoundException which is Checked	NoClassDefFoundError which is Unchecked
Object o = Class.forName(args[0]).newInstance(); java Test student ↵	Test t = new Test(); Hardcode the class-name

Eg.

Test t = new Test();

At runtime is Test.class not available

R.E. NoClassDefFoundError : Test

But

while using newInstance() method at runtime .class file not available then we will get

R.E. ClassNotFoundException.

Object o = Class.forName(args[0]).newInstance();

java Test Test123 ↵

At runtime, if Test123.class is not available then we will get

R.E. ClassNotFoundException : Test123

→ Senario (Real-Time) (newInstance())

/test1 → servlet → JSP → newInstance()

Java Based Web-App

Functionally Equivalent

Date / /
Page

(3.) instanceof (VS) isInstance()

→ instanceof is an operator in java, we can use instanceof to check whether the given object is of particular type or not and we know the type at begining.

```
Thread t = new Thread();
S.O.P(t instanceof Runnable);
```

↓ we write it at begining.

(But)

- isInstance() is a method present in j.l.Class, we can use this method to check whether the given object is of particular type or not and we don't know the type at begining & providing at runtime dynamically.

```
Class Test
{
  P S V main(String[] args) throws Exception
  {
    Thread t = new Thread();
    S.O.P(Class.forName(args[0]).isInstance(t));
  }
}
```

java Test Runnable (true) | java Test String (false)