Operators & Assignments

1.. Increment Decrement Operators

Increment Operator is used to increase the Variable Value while Decrement Operators are Used to Decrease the Variable Value.

Increment Operators are of two types:

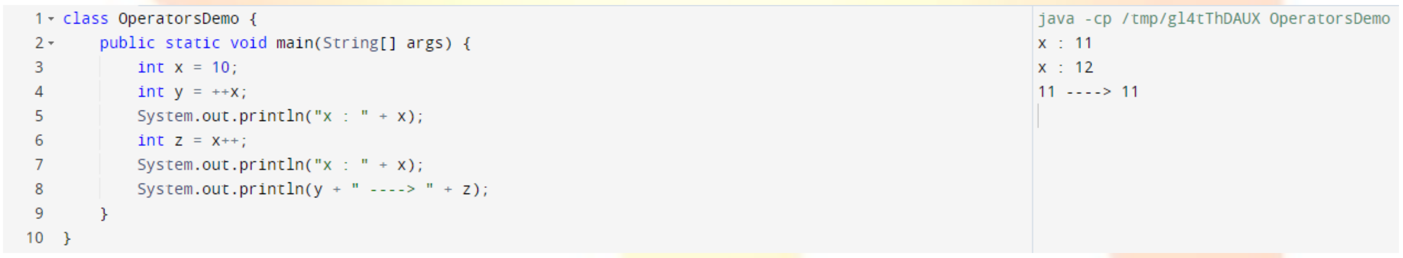
- Pre-Increment (y=++x)
- Post-Increment(y=x++)

Similarly, Decrement Operators are also of two types:

- Pre-Decrement (y=--x)
- Post-Decrement(y=x--)

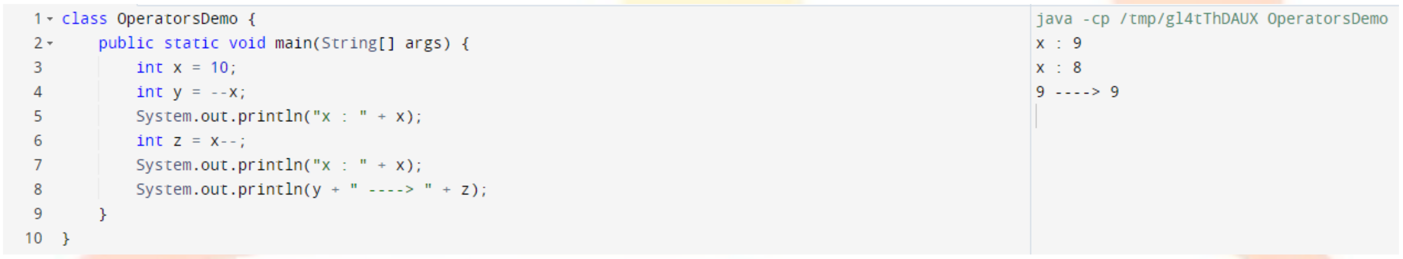
The following table will demonstrate the use of Increment and Decrement operators.

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



```java
class OperatorsDemo {
    public static void main(String[] args) {
        int x = 10;
        int y = ++x;
        System.out.println("x : " + x);
        int z = x++;
        System.out.println("x : " + x);
        System.out.println(y + " ----> " + z);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDemo
x : 11
x : 12
11 ----> 11
```



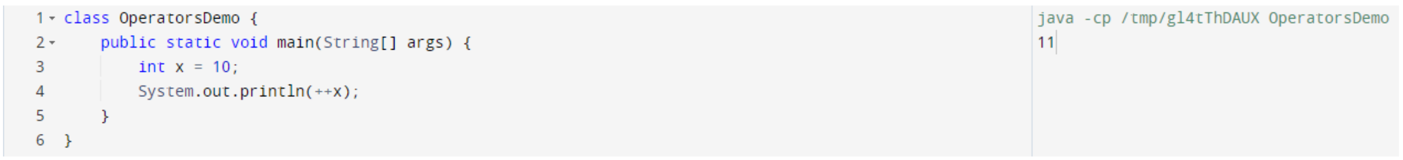
```java
class OperatorsDemo {
    public static void main(String[] args) {
        int x = 10;
        int y = --x;
        System.out.println("x : " + x);
        int z = x--;
        System.out.println("x : " + x);
        System.out.println(y + " ----> " + z);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDemo
x : 9
x : 8
9 ----> 9
```

Rules Related to Increment and Decrement Operators

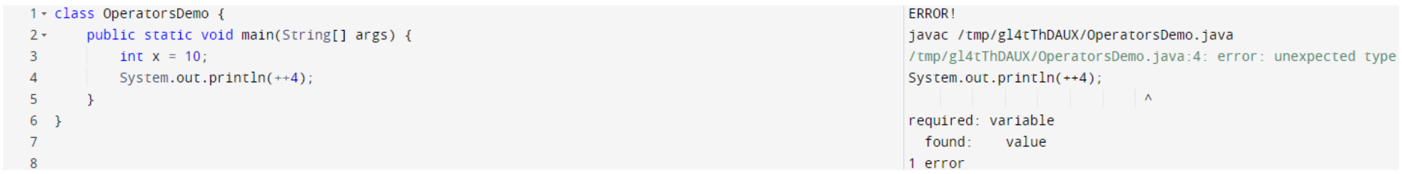
Rule – 1

Increment & decrement operators we can apply only for variables but not for constant values otherwise we will get compile time error.



```java
class OperatorsDemo {
    public static void main(String[] args) {
        int x = 10;
        System.out.println(++x);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDemo
11
```

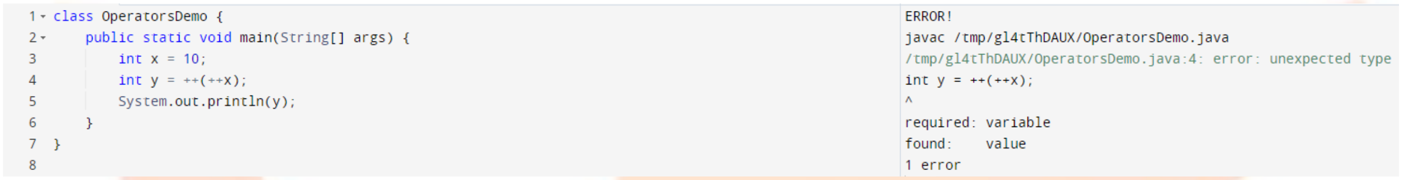


```java
class OperatorsDemo {
    public static void main(String[] args) {
        int x = 10;
        System.out.println(++4);
    }
}
```

```
ERROR!
javac /tmp/gl4tThDAUX/OperatorsDemo.java
/tmp/gl4tThDAUX/OperatorsDemo.java:4: error: unexpected type
System.out.println(++4);
                   ^
required: variable
  found:    value
1 error
```

Rule – 2

We can't perform nesting of increment or decrement operator, otherwise we will get compile time error.

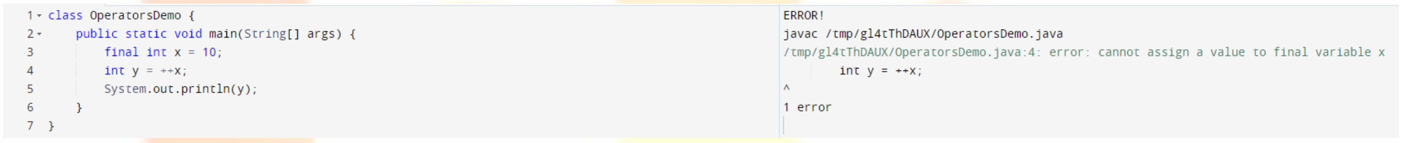


```java
class OperatorsDemo {
    public static void main(String[] args) {
        int x = 10;
        int y = ++(++x);
        System.out.println(y);
    }
}
```

```
ERROR!
javac /tmp/gl4tThDAUX/OperatorsDemo.java
/tmp/gl4tThDAUX/OperatorsDemo.java:4: error: unexpected type
int y = ++(++x);
^
required: variable
found:    value
1 error
```

Rule – 3

For the final variables we can't apply increment or decrement operators, otherwise we will get compile time error.

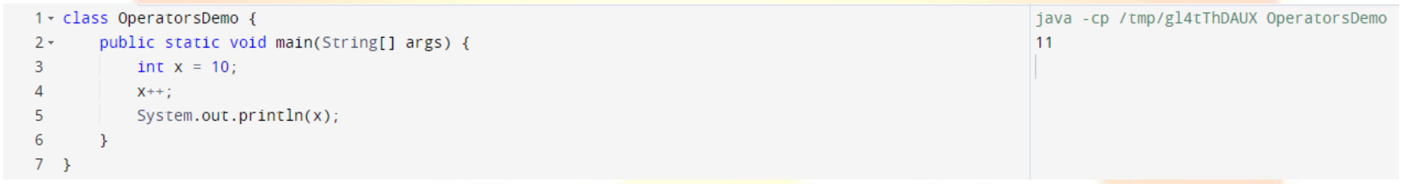


```java
class OperatorsDemo {
    public static void main(String[] args) {
        final int x = 10;
        int y = ++x;
        System.out.println(y);
    }
}
```

```
ERROR!
javac /tmp/gl4tThDAUX/OperatorsDemo.java
/tmp/gl4tThDAUX/OperatorsDemo.java:4: error: cannot assign a value to final variable x
        int y = ++x;
^
1 error
```

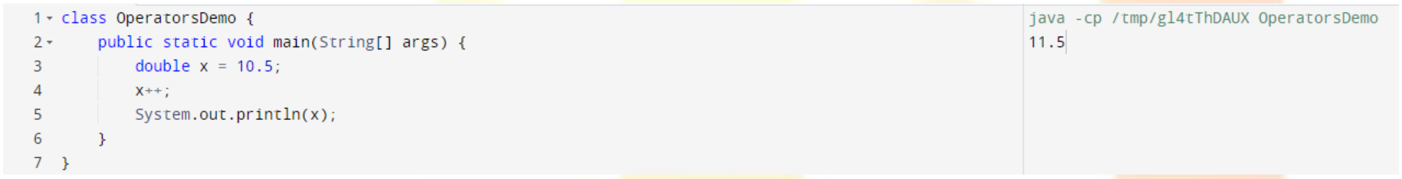
Rule – 4

We can apply increment or decrement operators even for primitive data types except boolean.



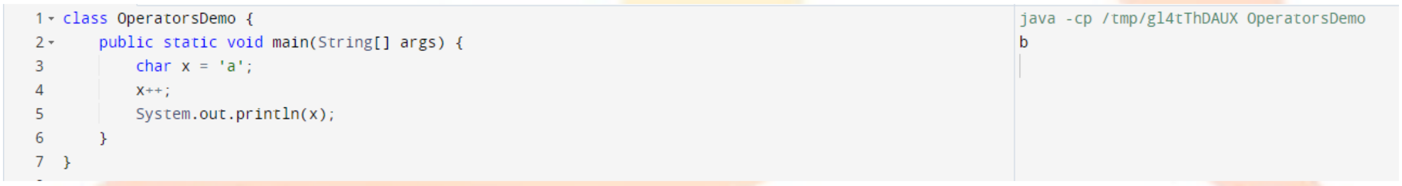
```java
class OperatorsDemo {
    public static void main(String[] args) {
        int x = 10;
        x++;
        System.out.println(x);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDemo
11
```



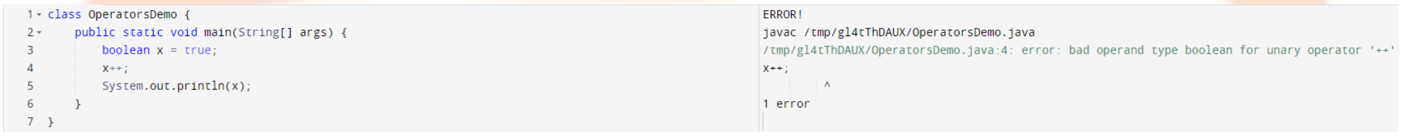
```java
class OperatorsDemo {
    public static void main(String[] args) {
        double x = 10.5;
        x++;
        System.out.println(x);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDemo
11.5
```



```java
class OperatorsDemo {
    public static void main(String[] args) {
        char x = 'a';
        x++;
        System.out.println(x);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDemo
b
```

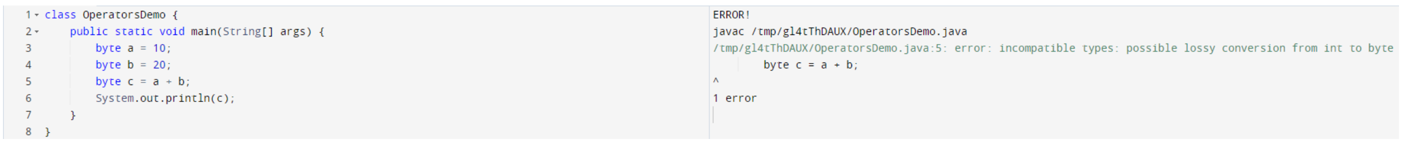


```java
class OperatorsDemo {
    public static void main(String[] args) {
        boolean x = true;
        x++;
        System.out.println(x);
    }
}
```

```
ERROR!
javac /tmp/gl4tThDAUX/OperatorsDemo.java
/tmp/gl4tThDAUX/OperatorsDemo.java:4: error: bad operand type boolean for unary operator '++'
x++;
       ^
1 error
```

Difference between b++ and b=b+1

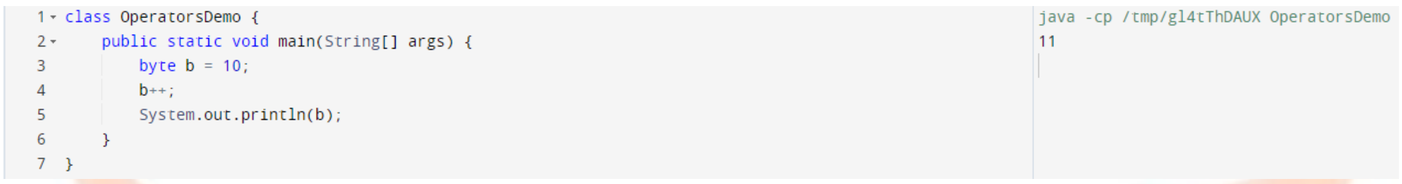
If we are applying any arithmetic operators between Two operands 'a' & 'b' the result type is max(int , type of a , type of b).



```java
class OperatorsDemo {
    public static void main(String[] args) {
        byte a = 10;
        byte b = 20;
        byte c = a + b;
        System.out.println(c);
    }
}
```

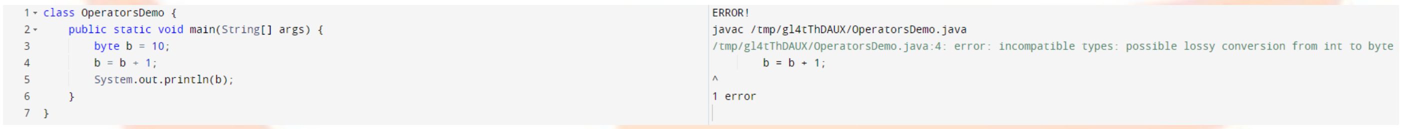
```
ERROR!
javac /tmp/gl4tThDAUX/OperatorsDemo.java
/tmp/gl4tThDAUX/OperatorsDemo.java:5: error: incompatible types: possible lossy conversion from int to byte
        byte c = a + b;
^
1 error
```

Coming to b++ and b=b+1, let's execute them and see.



```java
class OperatorsDemo {
    public static void main(String[] args) {
        byte b = 10;
        b++;
        System.out.println(b);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDemo
11
```



```java
class OperatorsDemo {
    public static void main(String[] args) {
        byte b = 10;
        b = b + 1;
        System.out.println(b);
    }
}
```

```
ERROR!
javac /tmp/gl4tThDAUX/OperatorsDemo.java
/tmp/gl4tThDAUX/OperatorsDemo.java:4: error: incompatible types: possible lossy conversion from int to byte
        b = b + 1;
^
1 error
```

In the case of Increment & Decrement operators internal type casting will be performed automatically by the compiler.

b++; => (type of b)(b+1); => (byte)(b+1);

2.. Arithmetic Operators

If we apply any Arithmetic operation (+,-,*,/,%) between two variables a & b, then the result type is always max(int , type of a , type of b).

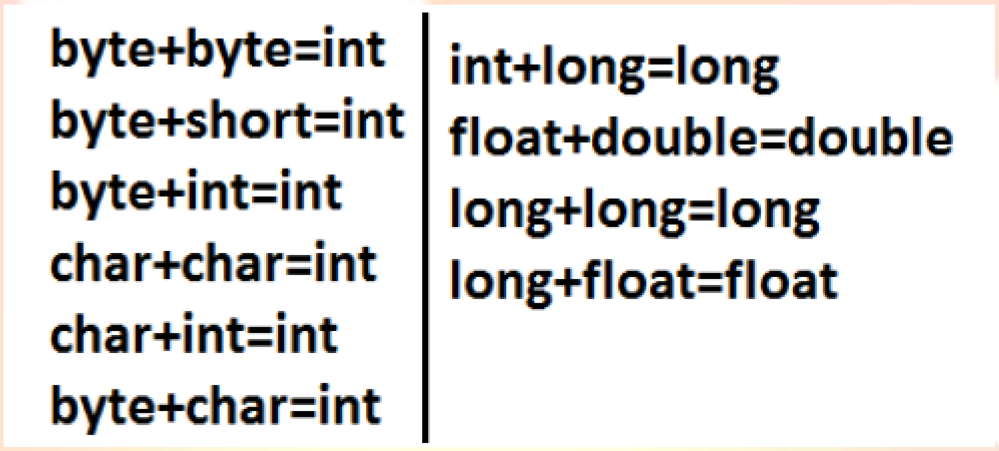
short + short=int

short + long=long

double + float=double

int + double=double

char + double=double

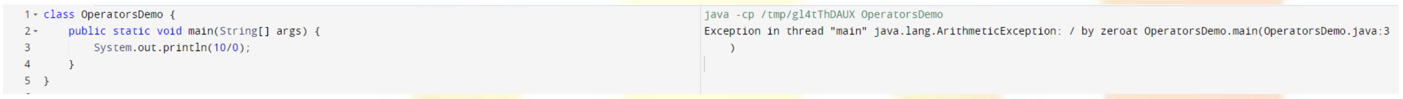


byte+byte=int	int+long=long
byte+short=int	float+double=double
byte+int=int	long+long=long
char+char=int	long+float=float
char+int=int	
byte+char=int	

Rules Related to Arithmetic Operators

Rule - 1

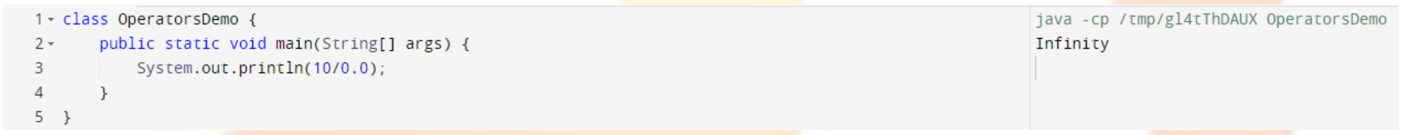
In integral arithmetic (byte, int, short, long) there is no way to represents infinity, if infinity is the result, we will get the ArithmeticException / by zero.



```java
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(10/0);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDemo
Exception in thread "main" java.lang.ArithmeticException: / by zeroat OperatorsDemo.main(OperatorsDemo.java:3
    )
```

While in floating point arithmetic(float, double) there is a way represents infinity.



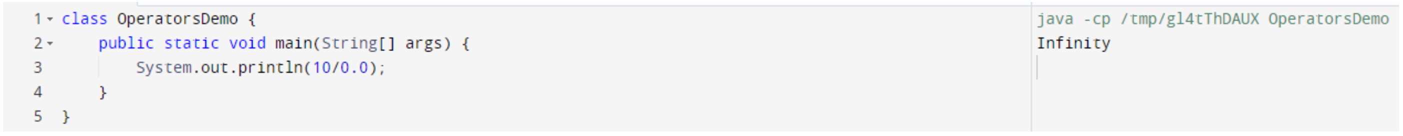
```java
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(10/0.0);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDemo
Infinity
```

For the Float & Double classes contains the following constants:

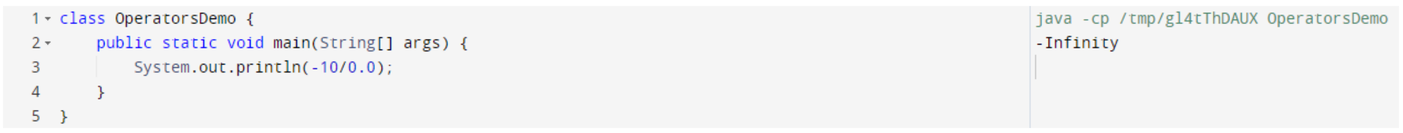
- POSITIVE_INFINITY
- NEGATIVE_INFINITY

Hence, if infinity is the result, we won't get any ArithmeticException in floating point arithmetic.



```java
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(10/0.0);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDemo
Infinity
```

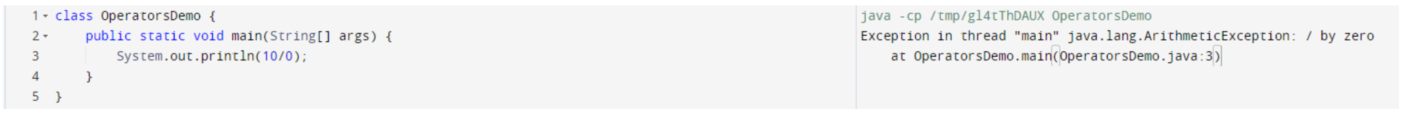


```java
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(-10/0.0);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDemo
-Infinity
```

Rule – 2

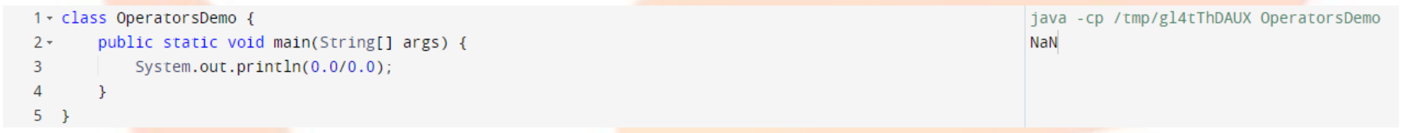
NaN (Not a Number) in integral arithmetic (byte, short, int, long) there is no way to represent undefine the results. Hence the result is undefined we will get ArithmeticException in integral arithmetic.



```
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(10/0);
    }
}
```

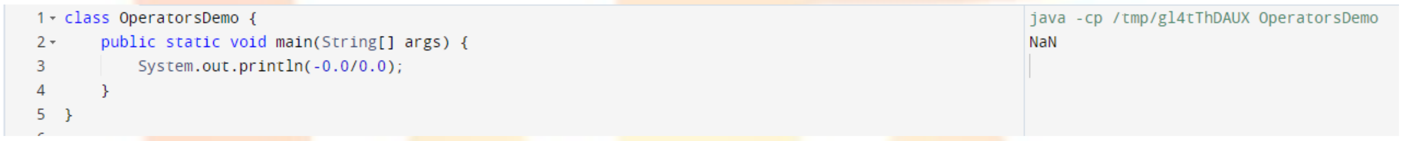
```
java -cp /tmp/gl4tThDAUX OperatorsDemo
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at OperatorsDemo.main(OperatorsDemo.java:3)
```

But floating point arithmetic (float, double) there is a way to represents undefined the results. For the Float , Double classes contains a constant NaN , Hence the result is undefined we won't get ArithmeticException in floating point arithmetics.



```
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(0.0/0.0);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDemo
NaN
```

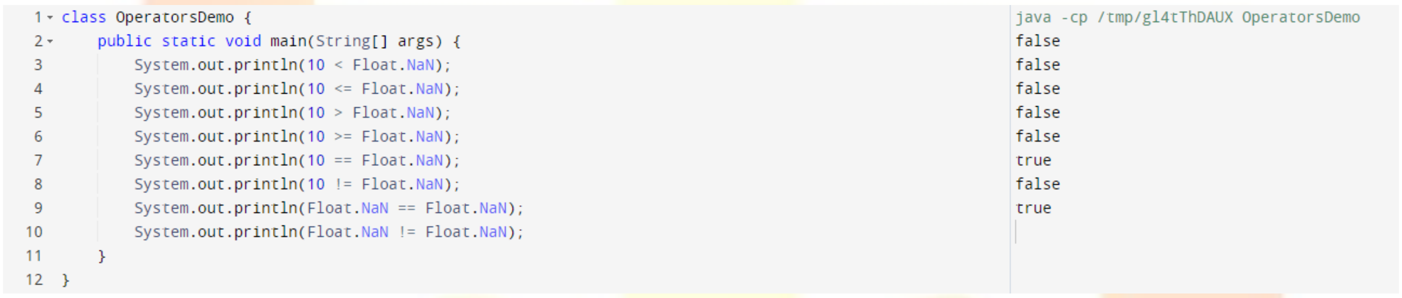


```
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(-0.0/0.0);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDemo
NaN
```

Rule – 3

For any 'x' value including NaN, the following expressions returns false.

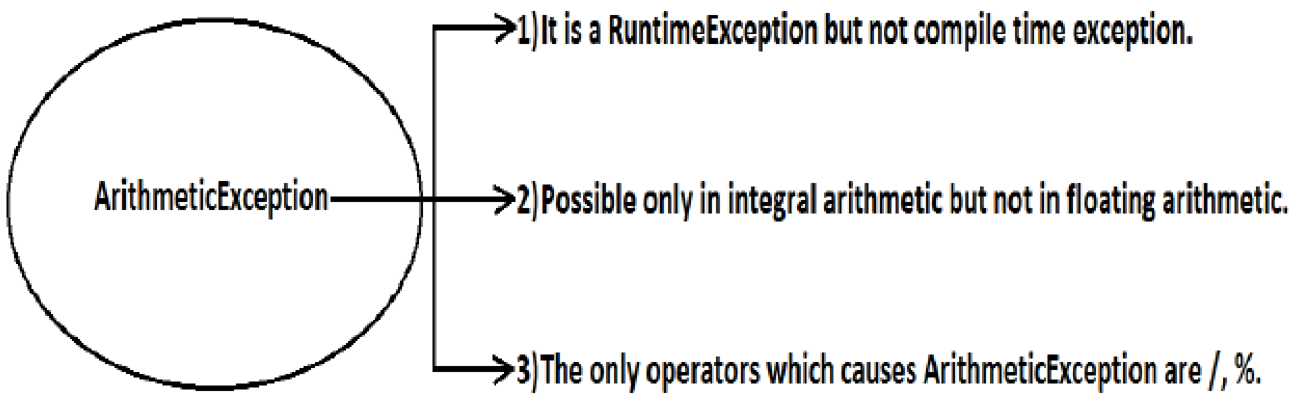


```
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(10 < Float.NaN);
        System.out.println(10 <= Float.NaN);
        System.out.println(10 > Float.NaN);
        System.out.println(10 >= Float.NaN);
        System.out.println(10 == Float.NaN);
        System.out.println(10 != Float.NaN);
        System.out.println(Float.NaN == Float.NaN);
        System.out.println(Float.NaN != Float.NaN);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDemo
false
false
false
false
false
true
false
true
```

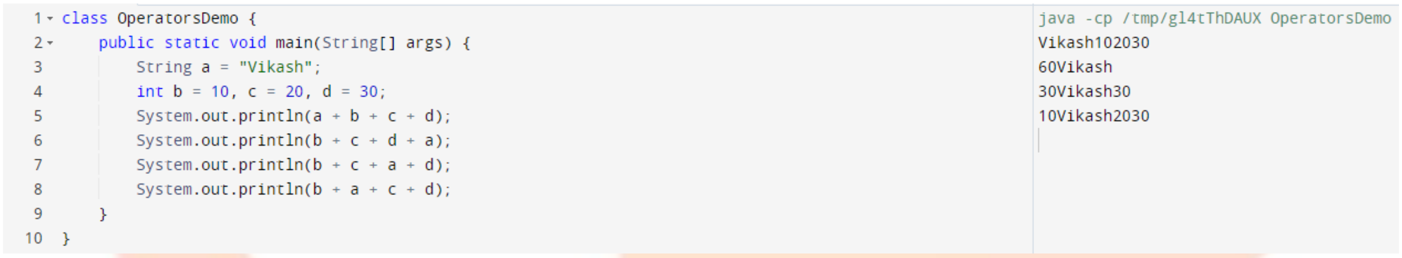
Rule – 4

ArithmeticException is a RuntimeException but not compile time error. It occurs only in integral arithmetic but not in floating point arithmetic. The only operations which cause ArithmeticException are: ' / ' and ' % '.


ArithmeticException
1)It is a RuntimeException but not compile time exception.
2)Possible only in integral arithmetic but not in floating arithmetic.
3)The only operators which causes ArithmeticException are /, %.

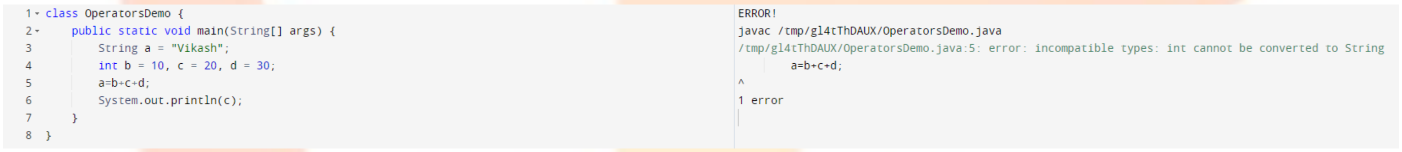
3.. String Concatenation Operators

The only overloaded operator in java is ' + ' operator sometimes it acts as an arithmetic addition operator and sometimes it acts as a **String concatenation** operator. If any of the one argument is of String type, then '+' operator acts as **String Concatenation Operator** and If both arguments are of number type, then operator acts as **Arithmetic Operator**.



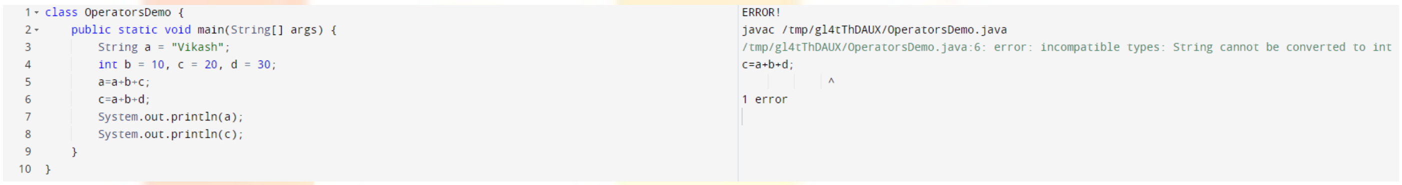
```java
class OperatorsDemo {
    public static void main(String[] args) {
        String a = "Vikash";
        int b = 10, c = 20, d = 30;
        System.out.println(a + b + c + d);
        System.out.println(b + c + d + a);
        System.out.println(b + c + a + d);
        System.out.println(b + a + c + d);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDemo
Vikash102030
60Vikash
30Vikash30
10Vikash2030
```



```java
class OperatorsDemo {
    public static void main(String[] args) {
        String a = "Vikash";
        int b = 10, c = 20, d = 30;
        a=b+c+d;
        System.out.println(c);
    }
}
```

```
ERROR!
javac /tmp/gl4tThDAUX/OperatorsDemo.java
/tmp/gl4tThDAUX/OperatorsDemo.java:5: error: incompatible types: int cannot be converted to String
        a=b+c+d;
^
1 error
```



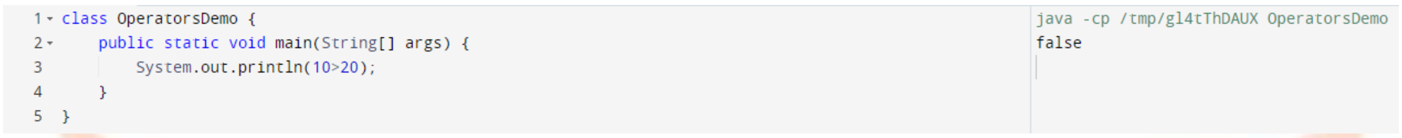
```java
class OperatorsDemo {
    public static void main(String[] args) {
        String a = "Vikash";
        int b = 10, c = 20, d = 30;
        a=a+b+c;
        c=a+b+d;
        System.out.println(a);
        System.out.println(c);
    }
}
```

```
ERROR!
javac /tmp/gl4tThDAUX/OperatorsDemo.java
/tmp/gl4tThDAUX/OperatorsDemo.java:6: error: incompatible types: String cannot be converted to int
c=a+b+d;
             ^
1 error
```

4.. Relational Operators (<, <=, >, >=)

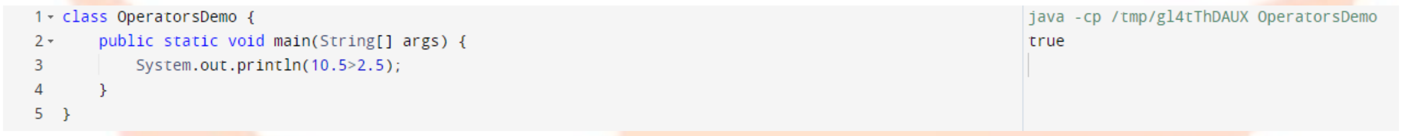
Relational Operators are used to compare two values whether the first value is less than, less than or equal to, greater than, greater than or equal to the second value.

We can apply relational operators for every primitive type except boolean.



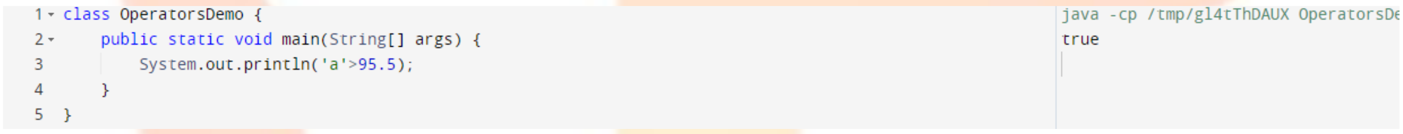
```java
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(10>20);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDemo
false
```



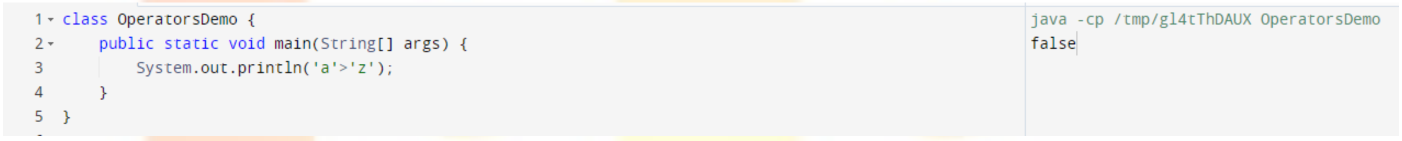
```java
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(10.5>2.5);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDemo
true
```



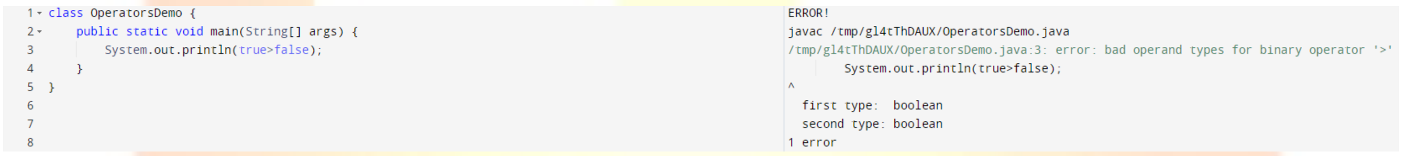
```java
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println('a'>95.5);
    }
}
```

```
java -cp /tmp/gl4tThDAUX OperatorsDe
true
```



```java
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println('a'>'z');
    }
}
```

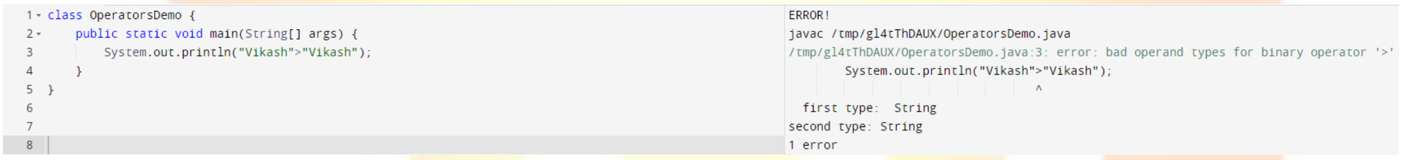
```
java -cp /tmp/gl4tThDAUX OperatorsDemo
false
```



```java
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(true>false);
    }
}
```

```
ERROR!
javac /tmp/gl4tThDAUX/OperatorsDemo.java
/tmp/gl4tThDAUX/OperatorsDemo.java:3: error: bad operand types for binary operator '>'
        System.out.println(true>false);
^
  first type:  boolean
  second type: boolean
1 error
```

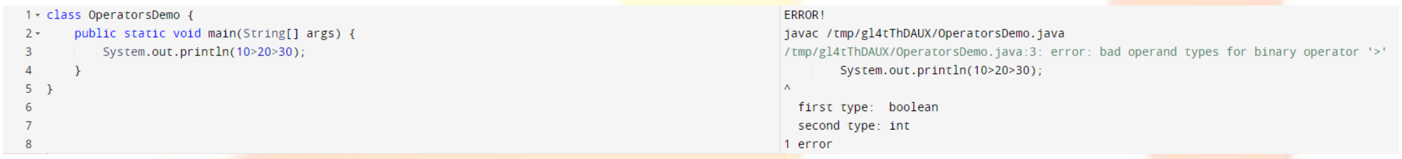
We can’t apply Relational Operators for Object Types.



```java
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println("Vikash">"Vikash");
    }
}
```

```
ERROR!
javac /tmp/gl4tThDAUX/OperatorsDemo.java
/tmp/gl4tThDAUX/OperatorsDemo.java:3: error: bad operand types for binary operator '>'
        System.out.println("Vikash">"Vikash");
                                   ^
  first type:  String
second type: String
1 error
```

Nesting of Relational Operators is Not Allowed.

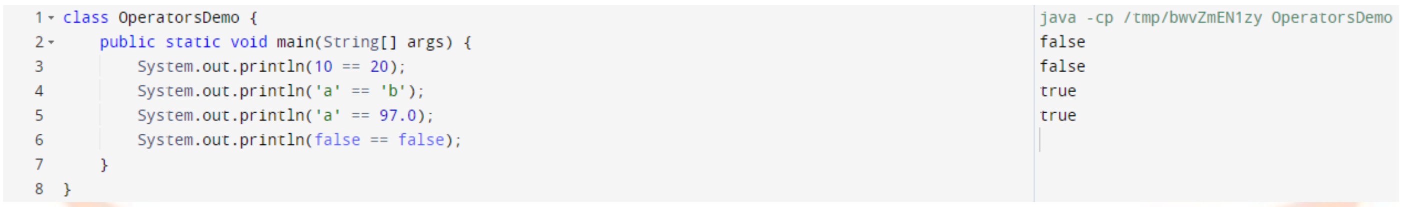


```java
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(10>20>30);
    }
}
```

```
ERROR!
javac /tmp/gl4tThDAUX/OperatorsDemo.java
/tmp/gl4tThDAUX/OperatorsDemo.java:3: error: bad operand types for binary operator '>'
        System.out.println(10>20>30);
^
  first type:  boolean
  second type: int
1 error
```

5.. Equality Operators (==, !=)

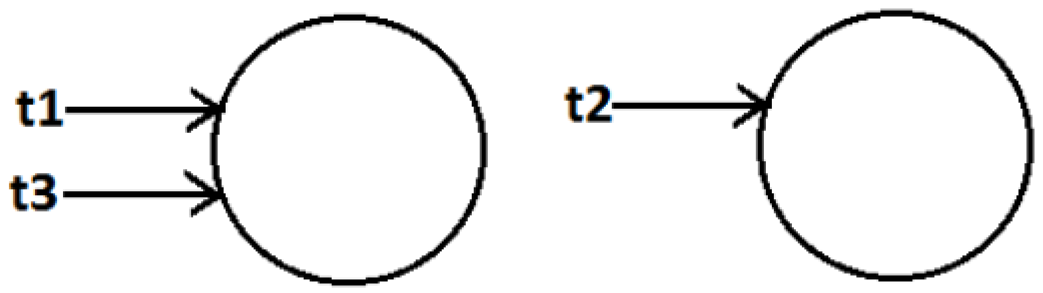
This Operators is used to check whether two values are equal or not equal to each other. We can apply equality operators for every primitive type including boolean type also.

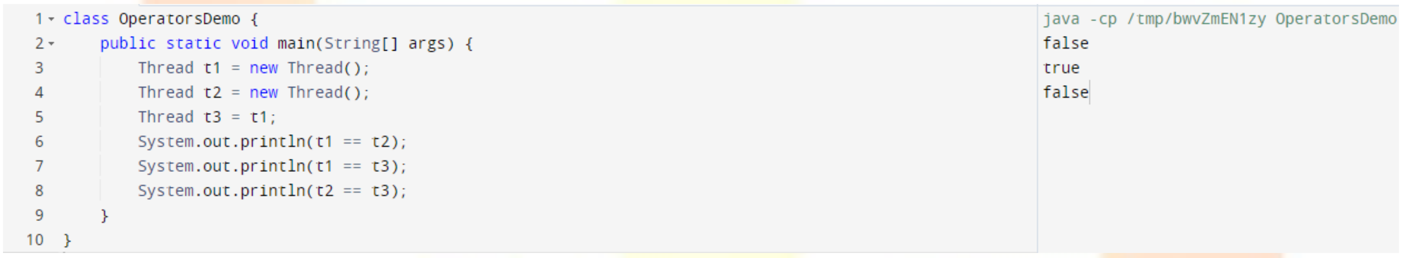


```java
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(10 == 20);
        System.out.println('a' == 'b');
        System.out.println('a' == 97.0);
        System.out.println(false == false);
    }
}
```

```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
false
false
true
true
```

We can apply equality operators for object types also. For object references r1 and r2, r1 == r2 returns true if and only if both r1 and r2 pointing to the same object. i.e., == operator meant for reference-comparison or address-comparison.

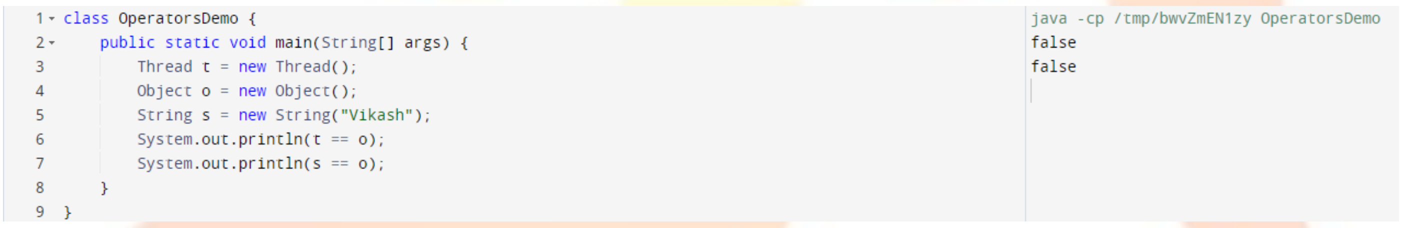




```java
class OperatorsDemo {
    public static void main(String[] args) {
        Thread t1 = new Thread();
        Thread t2 = new Thread();
        Thread t3 = t1;
        System.out.println(t1 == t2);
        System.out.println(t1 == t3);
        System.out.println(t2 == t3);
    }
}
```

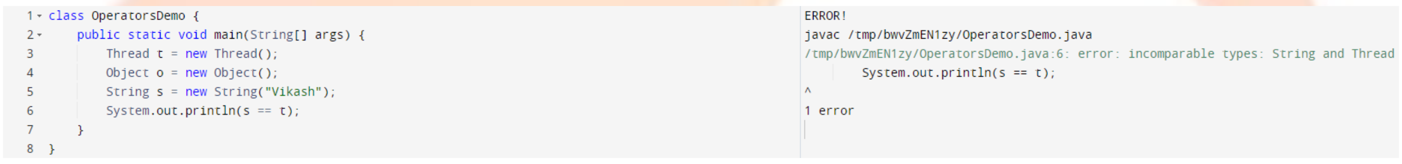
```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
false
true
false
```

To use the equality operators between object type compulsory these should be some relation between argument types (child to parent, parent to child), Otherwise we will get Compile time error incomparable types.



```java
class OperatorsDemo {
    public static void main(String[] args) {
        Thread t = new Thread();
        Object o = new Object();
        String s = new String("Vikash");
        System.out.println(t == o);
        System.out.println(s == o);
    }
}
```

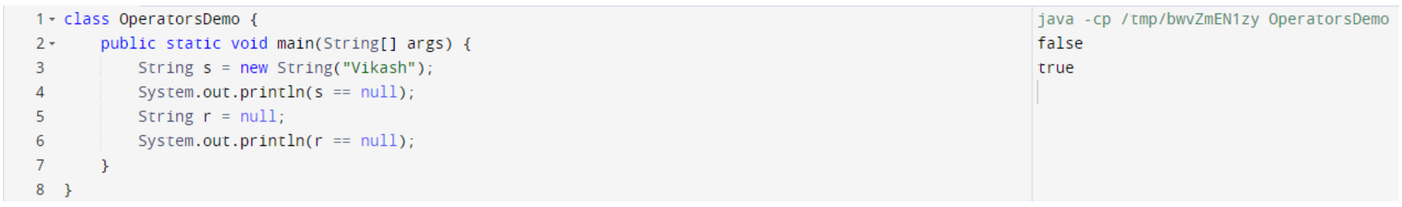
```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
false
false
```



```java
class OperatorsDemo {
    public static void main(String[] args) {
        Thread t = new Thread();
        Object o = new Object();
        String s = new String("Vikash");
        System.out.println(s == t);
    }
}
```

```
ERROR!
javac /tmp/bwvZmEN1zy/OperatorsDemo.java
/tmp/bwvZmEN1zy/OperatorsDemo.java:6: error: incomparable types: String and Thread
        System.out.println(s == t);
                             ^
1 error
```

For any object reference of on r==null is always false, but null==null is always true.

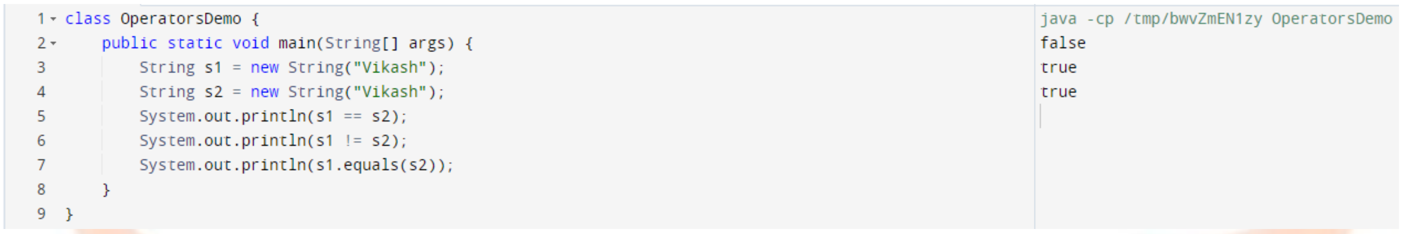


```java
class OperatorsDemo {
    public static void main(String[] args) {
        String s = new String("Vikash");
        System.out.println(s == null);
        String r = null;
        System.out.println(r == null);
    }
}
```

```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
false
true
```

Difference between == Operator and equals() method.

In general we can use equals( ) for content comparison whereas == operator for reference comparison.

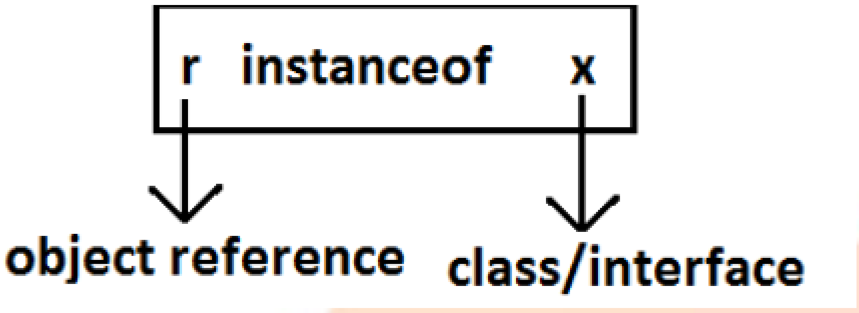


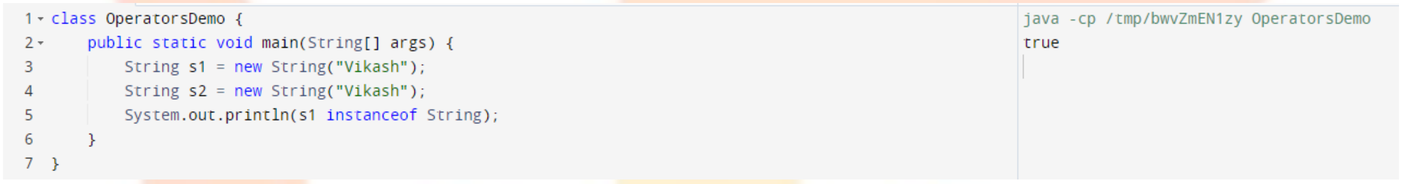
```java
class OperatorsDemo {
    public static void main(String[] args) {
        String s1 = new String("Vikash");
        String s2 = new String("Vikash");
        System.out.println(s1 == s2);
        System.out.println(s1 != s2);
        System.out.println(s1.equals(s2));
    }
}
```

```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
false
true
true
```

6.. instanceof Operators

We can use the instanceof operator to check whether the given an object is particular type or not.

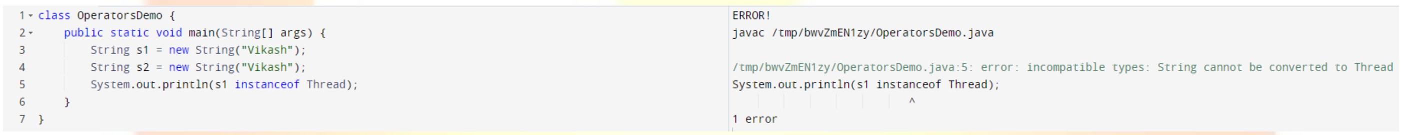




```java
class OperatorsDemo {
    public static void main(String[] args) {
        String s1 = new String("Vikash");
        String s2 = new String("Vikash");
        System.out.println(s1 instanceof String);
    }
}
```

```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
true
```

To use instance of operator compulsory there should be some relation between argument types (either child to parent or parent to child or same type) Otherwise we will get compile time error saying incompatible types.

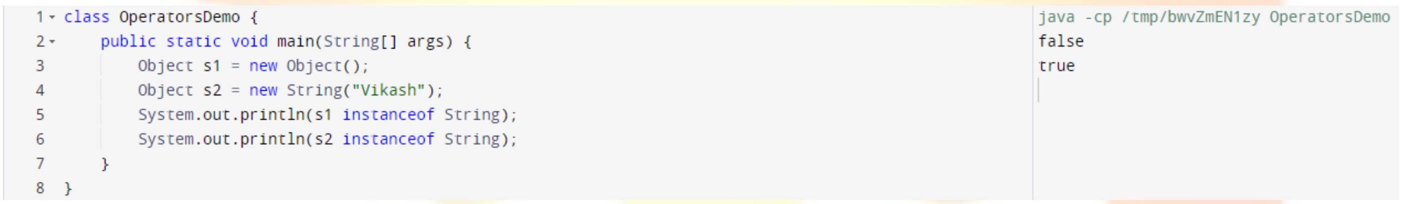


```java
class OperatorsDemo {
    public static void main(String[] args) {
        String s1 = new String("Vikash");
        String s2 = new String("Vikash");
        System.out.println(s1 instanceof Thread);
    }
}
```

```
ERROR!
javac /tmp/bwvZmEN1zy/OperatorsDemo.java

/tmp/bwvZmEN1zy/OperatorsDemo.java:5: error: incompatible types: String cannot be converted to Thread
System.out.println(s1 instanceof Thread);
                   ^
1 error
```

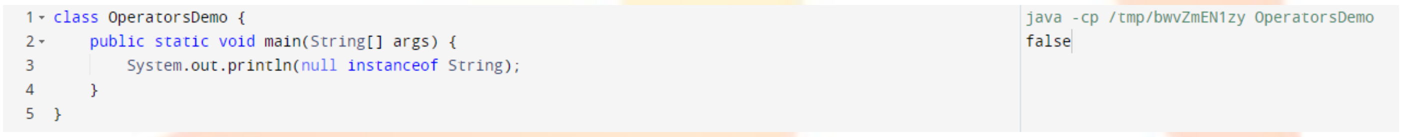
Whenever we are checking the parent object is child type or not by using instanceof operator that we get false.



```java
class OperatorsDemo {
    public static void main(String[] args) {
        Object s1 = new Object();
        Object s2 = new String("Vikash");
        System.out.println(s1 instanceof String);
        System.out.println(s2 instanceof String);
    }
}
```

```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
false
true
```

For any class or interface X null instanceof X is always returns false.

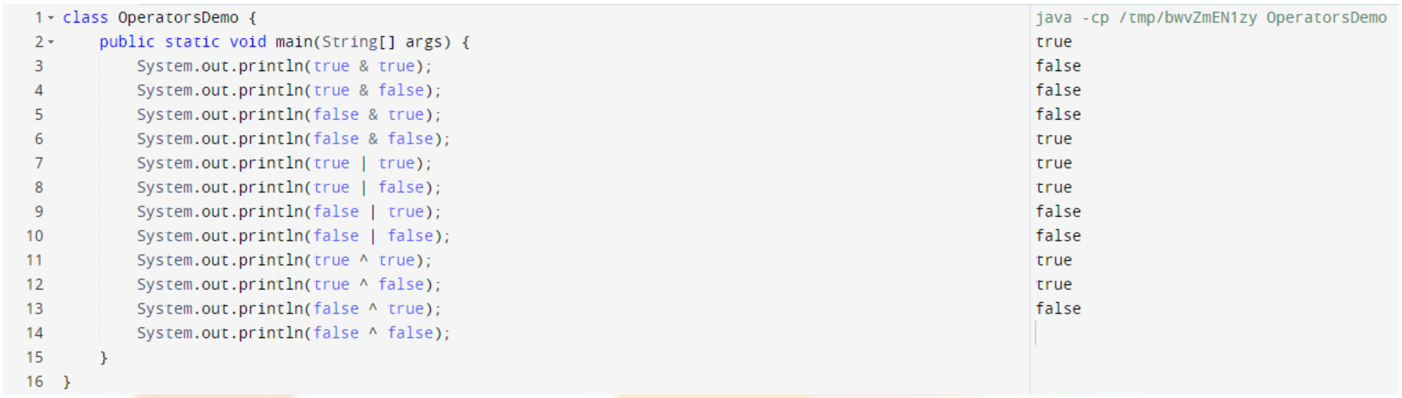


```java
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(null instanceof String);
    }
}
```

```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
false
```

7.. Bitwise Operators (&, |, ^)

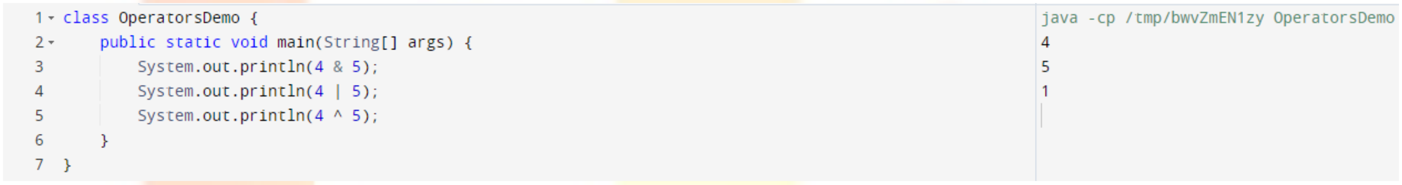
- & (AND): If both arguments are true then only result is true.
- | (OR): if at least one argument is true. Then the result is true.
- (X-OR): if both are different arguments. Then the result is true.



```
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(true & true);
        System.out.println(true & false);
        System.out.println(false & true);
        System.out.println(false & false);
        System.out.println(true | true);
        System.out.println(true | false);
        System.out.println(false | true);
        System.out.println(false | false);
        System.out.println(true ^ true);
        System.out.println(true ^ false);
        System.out.println(false ^ true);
        System.out.println(false ^ false);
    }
}
```

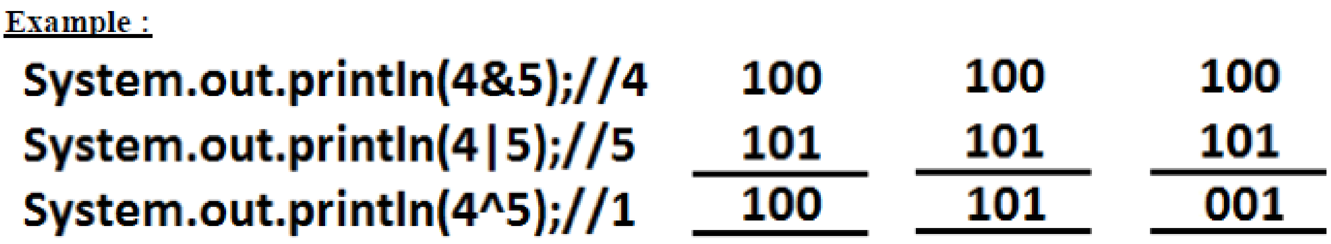
```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
true
false
false
false
true
true
true
false
false
true
true
false
```

We can apply bitwise operators even for integral types also.



```
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(4 & 5);
        System.out.println(4 | 5);
        System.out.println(4 ^ 5);
    }
}
```

```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
4
5
1
```

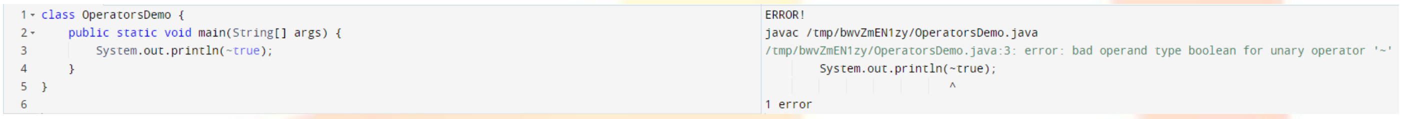


Example :

```
System.out.println(4&5);//4     100     100     100
System.out.println(4|5);//5     101     101     101
System.out.println(4^5);//1     ___     ___     ___
                                100     101     001
```

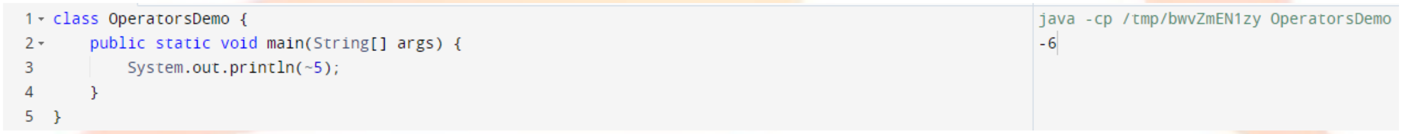
7.1 Bitwise Complement (~ Tilde) Operator

We can apply this operator only for integral types but not for boolean types.



```
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(~true);
    }
}
```

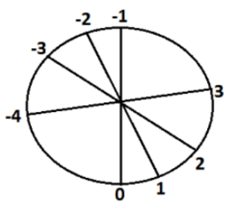
```
ERROR!
javac /tmp/bwvZmEN1zy/OperatorsDemo.java
/tmp/bwvZmEN1zy/OperatorsDemo.java:3: error: bad operand type boolean for unary operator '~'
        System.out.println(~true);
                           ^
1 error
```



```
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(~5);
    }
}
```

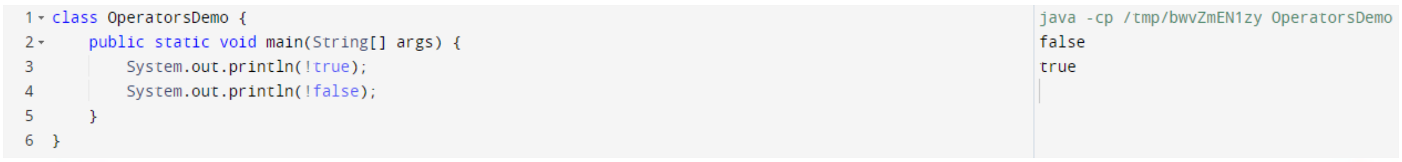
```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
-6
```

Note: The most significant bit access as sign bit 0 means positive number, 1 means negative number. Positive number will be represented directly in memory whereas negative number will be represented in 2's complement form.



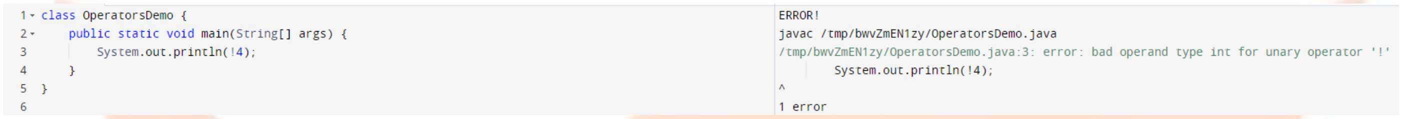
8.. Logical Operators (&&, ||, !)

Boolean Complement Operator (!) cannot be applied to Integral Types it can be applied to only Boolean Types.



```java
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(!true);
        System.out.println(!false);
    }
}
```

```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
false
true
```



```java
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(!4);
    }
}

```

```
ERROR!
javac /tmp/bwvZmEN1zy/OperatorsDemo.java
/tmp/bwvZmEN1zy/OperatorsDemo.java:3: error: bad operand type int for unary operator '!'
        System.out.println(!4);
^
1 error
```

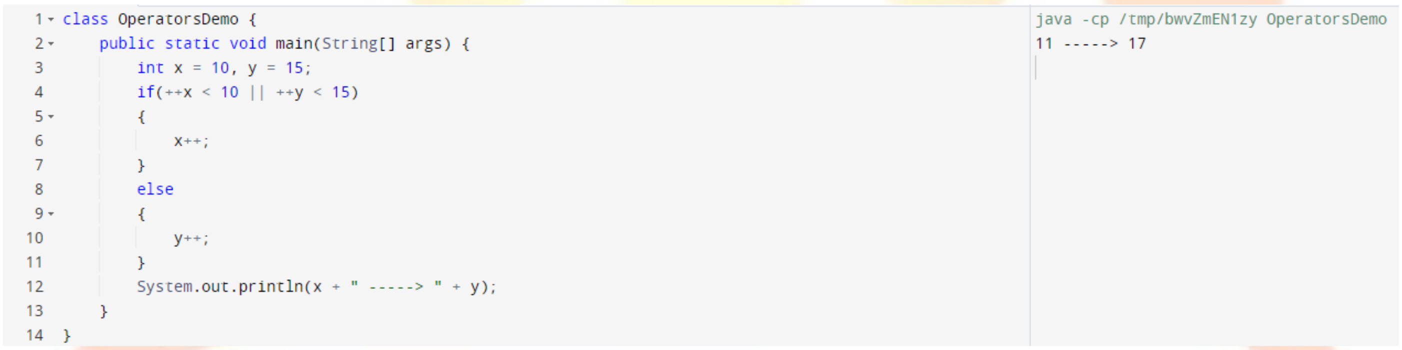
Logical AND (&&) and Logical OR (||) are also called **Short Circuit Operators** and these are same as Bitwise AND (&) and Bitwise OR (|) except the difference that these are applicable to only Boolean Types and here second argument will be evaluated only if first argument doesn’t match to fit i.e., Second argument evaluation is optional here while in Bitwise Operators Second Argument evaluation is Mandatory.

x&&y: y will be evaluated if and only if x is true. (If x is false then y won't be evaluated i.e., If x is true then only y will be evaluated)

x||y: y will be evaluated if and only if x is false. (If x is true then y won't be evaluated i.e., If x is false then only y will be evaluated)

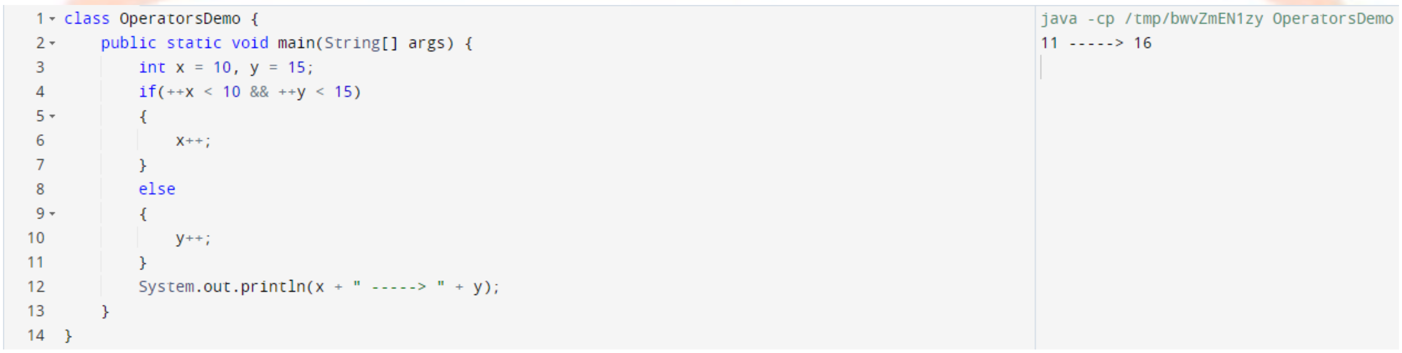
Summary:

- & -> Applicable for both boolean and integral types.
- | -> Applicable for both boolean and integral types.
- ^ -> Applicable for both boolean and integral types.
- ~ -> Applicable for integral types only but not for boolean types.
- ! -> Applicable for boolean types only but not for integral types.
- && -> Applicable for boolean types only but not for integral types.
- || -> Applicable for boolean types only but not for integral types.



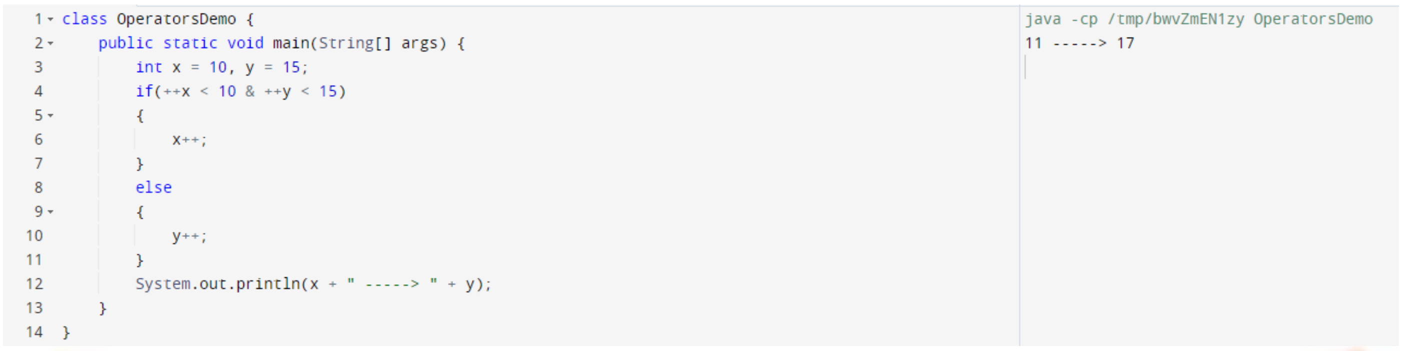
```java
class OperatorsDemo {
    public static void main(String[] args) {
        int x = 10, y = 15;
        if(++x < 10 || ++y < 15)
        {
            x++;
        }
        else
        {
            y++;
        }
        System.out.println(x + " -----> " + y);
    }
}
```

```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
11 -----> 17
```



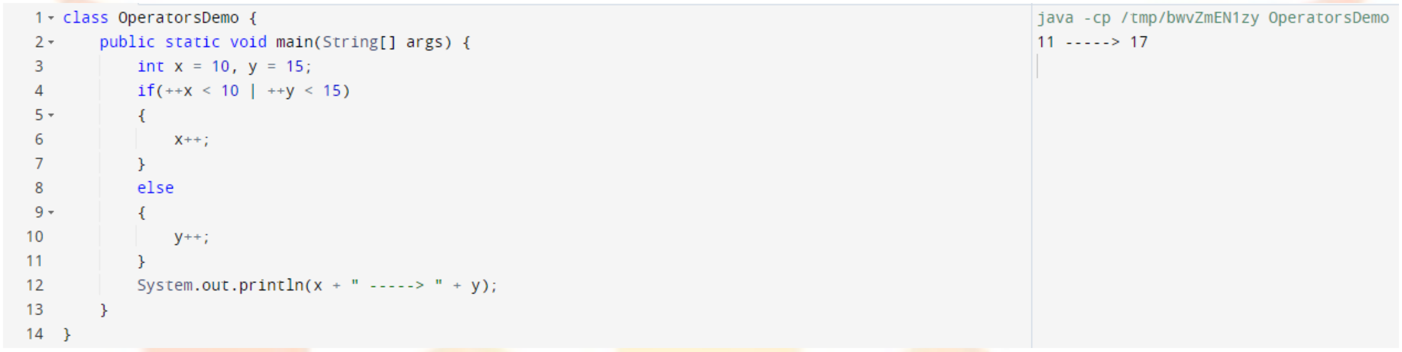
```java
class OperatorsDemo {
    public static void main(String[] args) {
        int x = 10, y = 15;
        if(++x < 10 && ++y < 15)
        {
            x++;
        }
        else
        {
            y++;
        }
        System.out.println(x + " -----> " + y);
    }
}
```

```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
11 -----> 16
```



```java
class OperatorsDemo {
    public static void main(String[] args) {
        int x = 10, y = 15;
        if(++x < 10 & ++y < 15)
        {
            x++;
        }
        else
        {
            y++;
        }
        System.out.println(x + " -----> " + y);
    }
}
```

```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
11 -----> 17
```



```java
class OperatorsDemo {
    public static void main(String[] args) {
        int x = 10, y = 15;
        if(++x < 10 | ++y < 15)
        {
            x++;
        }
        else
        {
            y++;
        }
        System.out.println(x + " -----> " + y);
    }
}
```

```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
11 -----> 17
```

9.. Type Cast Operators

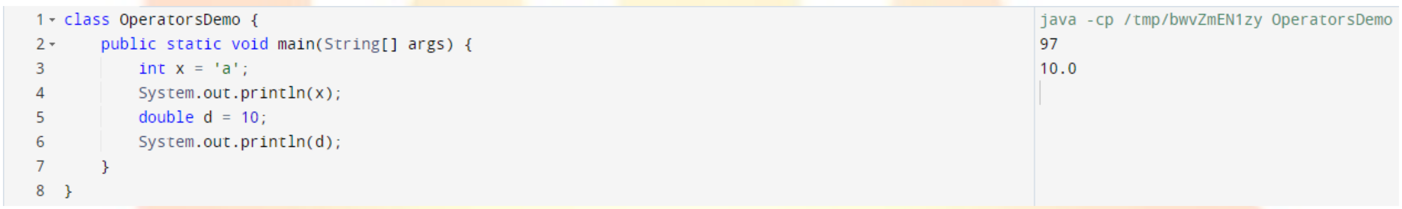
There are 2 types of type-casting

- Implicit Type Casting
- Explicit Type Casting

9.1 Implicit Type Casting

- The compiler is responsible to perform this type casting.
- Whenever we are assigning lower datatype value to higher datatype variable then implicit type cast will be performed.
- It is also known as Widening or Upcasting.
- There is no loss of information in this type casting.
- The following are various possible implicit type casting.

$$\begin{tikzcd}
\text{byte} \arrow[r] & \text{short} \arrow[dr] & & & & \\
& & \text{int} \arrow[r] & \text{long} \arrow[r] & \text{float} \arrow[r] & \text{double} \\
& \text{char} \arrow[ur] & & & &
\end{tikzcd}$$



```
class OperatorsDemo {
    public static void main(String[] args) {
        int x = 'a';
        System.out.println(x);
        double d = 10;
        System.out.println(d);
    }
}
```

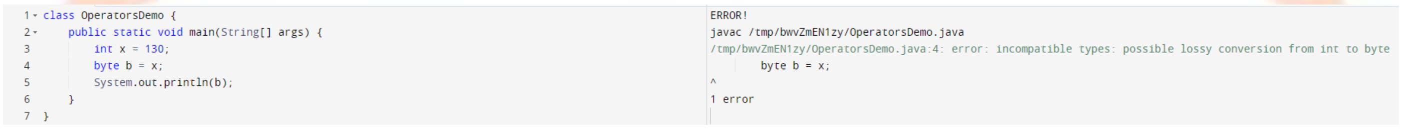
```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
97
10.0
```

Note: Compiler converts char to int type and int to double type automatically by implicit type casting.

9.2 Explicit Type Casting

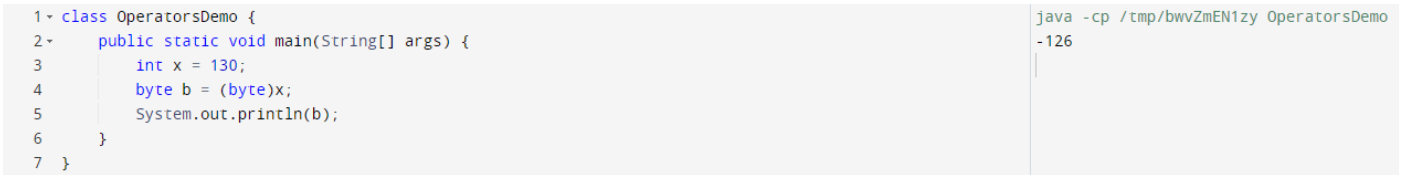
- Programmer is responsible for this type casting.
- Whenever we are assigning bigger data type value to the smaller data type variable then explicit type casting is required.
- Also known as Narrowing or down casting.
- There may be a chance of loss of information in this type casting.
- The following are various possible conversions where explicit type casting is required.

$$\begin{tikzcd}
\begin{array}{c}\text{byte}\\8\end{array} \arrow[r] & \begin{array}{c}\text{short}\\16\end{array} \arrow[dr] & & & & \\
& & \begin{array}{c}\text{int}\\32\end{array} \arrow[r] & \begin{array}{c}\text{long}\\64\end{array} \arrow[r] & \begin{array}{c}\text{float}\\32\end{array} \arrow[r] & \text{double} \\
& \begin{array}{c}\text{char}\\16\end{array} \arrow[ur] & & & &
\end{tikzcd}$$



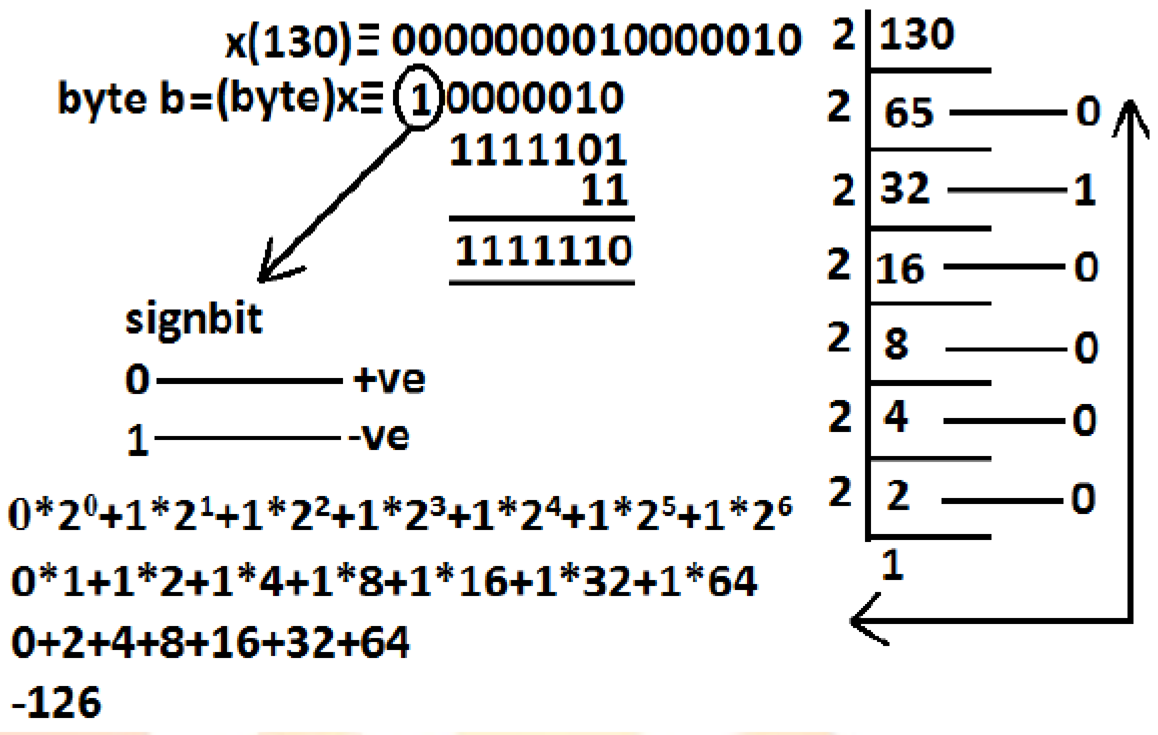
```
class OperatorsDemo {
    public static void main(String[] args) {
        int x = 130;
        byte b = x;
        System.out.println(b);
    }
}
```

```
ERROR!
javac /tmp/bwvZmEN1zy/OperatorsDemo.java
/tmp/bwvZmEN1zy/OperatorsDemo.java:4: error: incompatible types: possible lossy conversion from int to byte
        byte b = x;
^
1 error
```

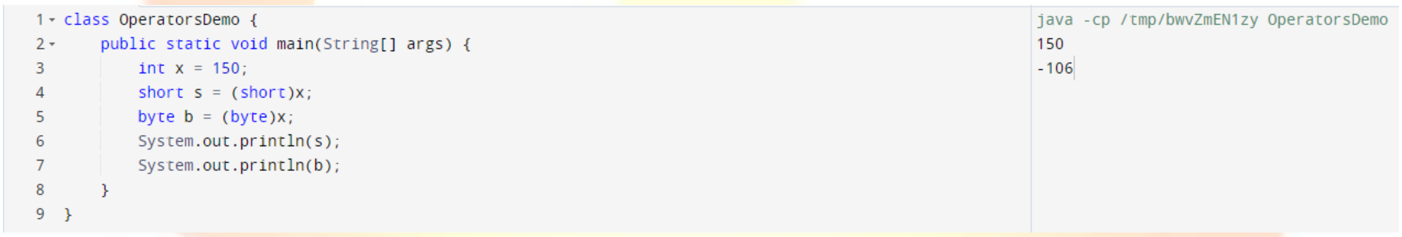


```
class OperatorsDemo {
    public static void main(String[] args) {
        int x = 130;
        byte b = (byte)x;
        System.out.println(b);
    }
}
```

```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
-126
```



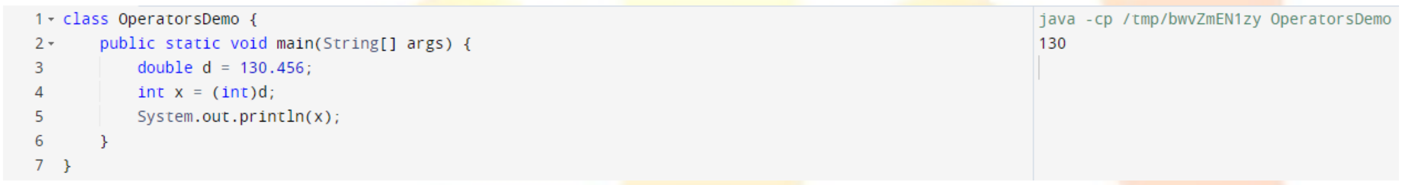
Whenever we are assigning higher datatype value to lower datatype value variable by explicit type-casting, the most significant bits will be lost i.e., we have considered least significant bits.



```
class OperatorsDemo {
    public static void main(String[] args) {
        int x = 150;
        short s = (short)x;
        byte b = (byte)x;
        System.out.println(s);
        System.out.println(b);
    }
}
```

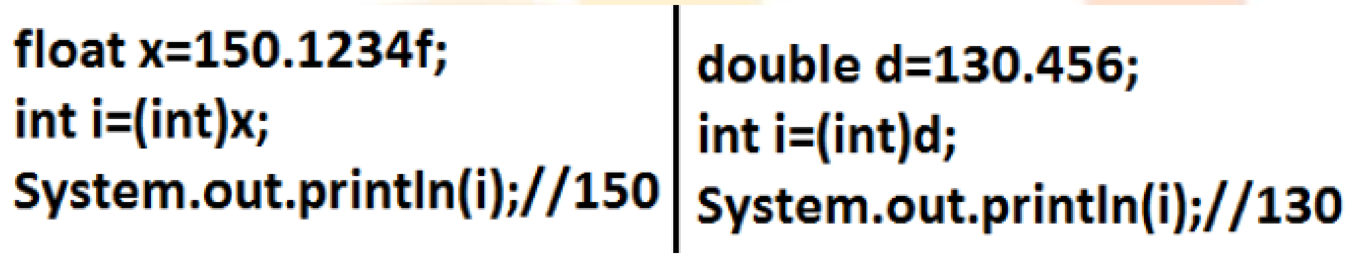
```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
150
-106
```

Whenever we are assigning floating point value to the integral types by explicit Type-casting, the digits of after decimal point will be lost.



```
class OperatorsDemo {
    public static void main(String[] args) {
        double d = 130.456;
        int x = (int)d;
        System.out.println(x);
    }
}
```

```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
130
```



```
float x=150.1234f;
int i=(int)x;
System.out.println(i);//150
```

```
double d=130.456;
int i=(int)d;
System.out.println(i);//130
```

10.. Assignment Operators

There are Three types of assignment operators

- Simple Assignment
- Chained Assignment
- Compound Assignment

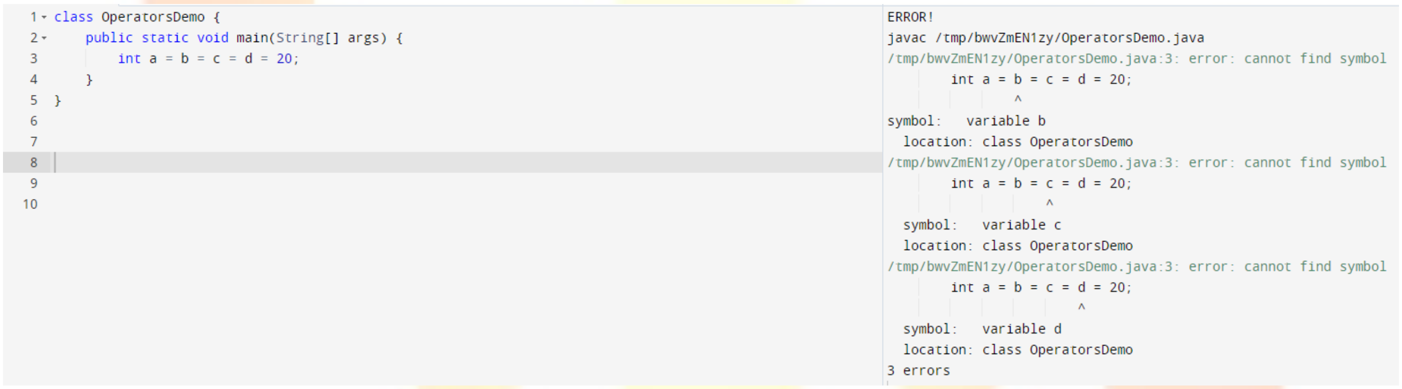
Simple Assignment

```java
class OperatorsDemo {
    public static void main(String[] args) {
        int x = 10;
    }
}
```

Chained Assignment

```java
class OperatorsDemo {
    public static void main(String[] args) {
        int a,b,c,d;
        a = b = c = d = 20;
    }
}
```

We can't perform chained assignment directly at the time of declaration.

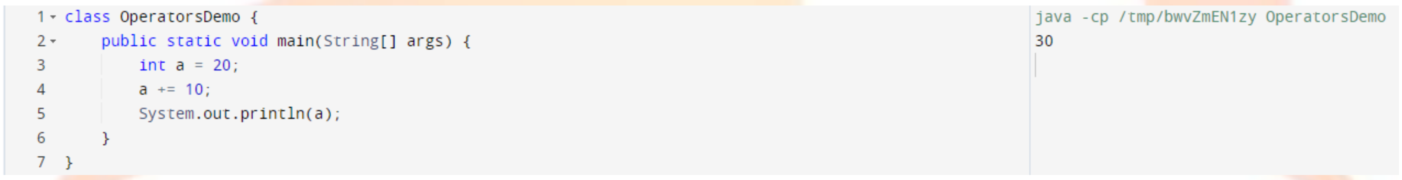


```java
class OperatorsDemo {
    public static void main(String[] args) {
        int a = b = c = d = 20;
    }
}
```

```
ERROR!
javac /tmp/bwvZmEN1zy/OperatorsDemo.java
/tmp/bwvZmEN1zy/OperatorsDemo.java:3: error: cannot find symbol
        int a = b = c = d = 20;
                ^
symbol:   variable b
  location: class OperatorsDemo
/tmp/bwvZmEN1zy/OperatorsDemo.java:3: error: cannot find symbol
        int a = b = c = d = 20;
                    ^
  symbol:   variable c
  location: class OperatorsDemo
/tmp/bwvZmEN1zy/OperatorsDemo.java:3: error: cannot find symbol
        int a = b = c = d = 20;
                        ^
  symbol:   variable d
  location: class OperatorsDemo
3 errors
```

Compound Assignment

Sometimes we can mix assignment operator with some other operator to form compound assignment operator.



```java
class OperatorsDemo {
    public static void main(String[] args) {
        int a = 20;
        a += 10;
        System.out.println(a);
    }
}
```

```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
30
```

The following is the list of all possible compound assignment operators in java.

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

In the case of compound assignment operator internal type casting will be performed automatically by the compiler (similar to increment and decrement operators.)

```
byte b=10;
b=b+1; ----C.E---->
System.out.println(b);
```

```
E:\scjp>javac OperatorsDemo.java
OperatorsDemo.java:6: possible loss of precision
found   : int
required: byte
            b=b+1;
```

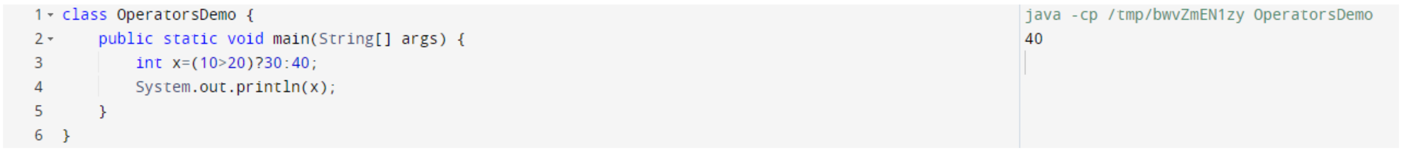
```
byte b=10;
b++;
System.out.println(b);//11
```

```
byte b=10;
//b+=1;
b=(byte)(b+1);
System.out.println(b);//11
```

```
int a,b,c,d;
a=b=c=d=20;
a+=b-=c*=d/=2;
System.out.println(a+"--"+b+"---"+c+"---"+d);
                    //-160---180---200---10
```

11.. Conditional Operators (?:)

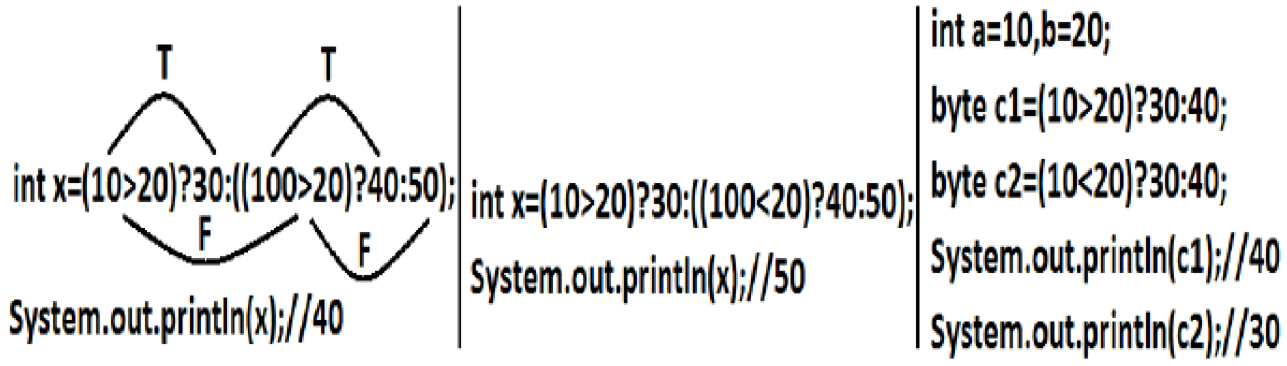
The only possible ternary operator in java is conditional operator.



```
class OperatorsDemo {
    public static void main(String[] args) {
        int x=(10>20)?30:40;
        System.out.println(x);
    }
}
```

```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
40
```

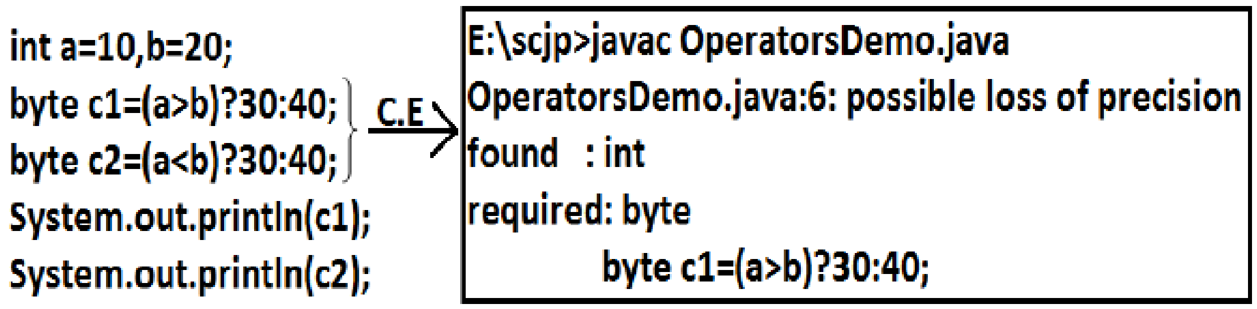
Nesting of conditional operator is possible.



```
int x=(10>20)?30:((100>20)?40:50);
System.out.println(x);//40
```

```
int x=(10>20)?30:((100<20)?40:50);
System.out.println(x);//50
```

```
int a=10,b=20;
byte c1=(10>20)?30:40;
byte c2=(10<20)?30:40;
System.out.println(c1);//40
System.out.println(c2);//30
```



```
int a=10,b=20;
byte c1=(a>b)?30:40;
byte c2=(a<b)?30:40;
System.out.println(c1);
System.out.println(c2);
```

```
E:\scjp>javac OperatorsDemo.java
OperatorsDemo.java:6: possible loss of precision
found   : int
required: byte
          byte c1=(a>b)?30:40;
```

12.. new Operator

- We can use "new" operator to create an object.
- There is no "delete" operator in java because destruction of useless objects is the responsibility of garbage collector.

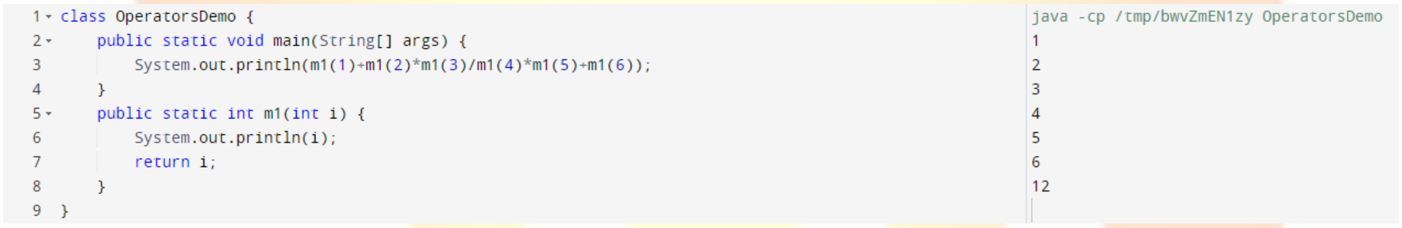
13.. [] Operator

We can use this operator to declare under construct/create arrays.

14.. Java Operator Precedence

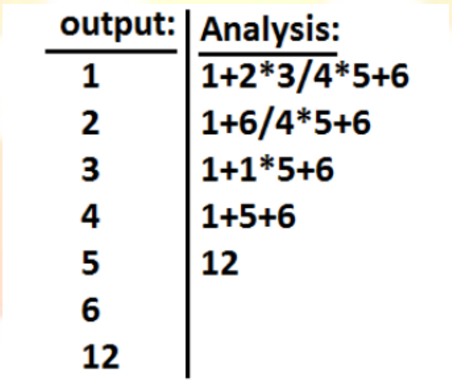
- Unary operators: [], x++, x--, ++x, --x, ~, !, new, <type>
- Arithmetic operators: *, /, %, +, -
- Shift operators: >>, >>>, <<
- Comparision operators: <, <=, >, >=, instanceof.
- Equality operators: ==, !=
- Bitwise operators: &, ^, |
- Short circuit operators: &&, ||
- Conditional operator: (?:)
- Assignment operators: +=, -=, *=, /=, %= . . .

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

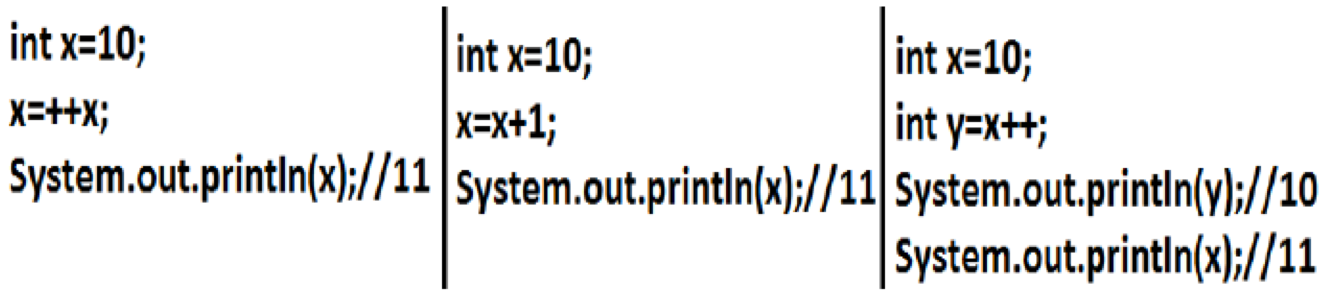


```java
class OperatorsDemo {
    public static void main(String[] args) {
        System.out.println(m1(1)+m1(2)*m1(3)/m1(4)*m1(5)+m1(6));
    }
    public static int m1(int i) {
        System.out.println(i);
        return i;
    }
}
```

```
java -cp /tmp/bwvZmEN1zy OperatorsDemo
1
2
3
4
5
6
12
```



output:	Analysis:
1	1+2*3/4*5+6
2	1+6/4*5+6
3	1+1*5+6
4	1+5+6
5	12
6	
12	



```java
int x=10;
x=++x;
System.out.println(x);//11
```

```java
int x=10;
x=x+1;
System.out.println(x);//11
```

```java
int x=10;
int y=x++;
System.out.println(y);//10
System.out.println(x);//11
```

15.. new vs newInstance()

new is an operator to create an objects, if we know class name at the beginning then we can create an object by using new operator. On the other hand newInstance( ) is a method presenting class " Class " , which can be used to create object. If we don't know the class name at the beginning and its available dynamically Runtime then we should go for newInstance() method.

If dynamically provide class name is not available then we will get the RuntimeException saying ClassNotFoundException. To use newInstance( ) method compulsory corresponding class should contains no argument constructor , otherwise we will get the RuntimeException saying InstantiationException.

new	newInstance( )
new is an operator , which can be used to create an object	newInstance( ) is a method , present in class Class , which can be used to create an object .
We can use new operator if we know the class name at the beginning. Test t= new Test( );	We can use the newInstance( ) method , If we don't class name at the beginning and available dynamically Runtime. Object o=Class.forName(arg[0]).newInstance( );
If the corresponding .class file not available at Runtime then we will get RuntimeException saying NoClassDefFoundError , It is unchecked	If the corresponding .class file not available at Runtime then we will get RuntimeException saying ClassNotFoundException , It is checked
To used new operator the corresponding class not required to contain no argument constructor	To used newInstance( ) method the corresponding class should compulsory contain no argument constructor , Other wise we will get RuntimeException saying InstantiationException.

16.. Difference between ClassNotFoundException & NoClassDefFoundError

For hard coded class names at Runtime in the corresponding .class files not available we will get **NoClassDefFoundError**, which is unchecked. Consider an example:

Test t = new Test( );

In Runtime if Test.class file is not available then we will get NoClassDefFoundError.

For Dynamically provided class names at Runtime , If the corresponding .class files is not available then we will get the RuntimeException saying **ClassNotFoundException**

Example:

Object o=Class.forname("Test").newInstance( );

At Runtime if Test.class file not available then we will get the **ClassNotFoundException**, which is checked exception.

17.. Difference between instanceof and isInstance()

instanceof	isInstance( )
instanceof an operator which can be used to check whether the given object is perticular type or not We know at the type at beginning it is available	isInstance( ) is a method , present in class Class , we can use isInstance( ) method to checked whether the given object is perticular type or not We don't know at the type at beginning it is available Dynamically at Runtime.

`String  s = new String("ashok");`
`System.out.println(s instanceof Object );`
 	
`//true`
If we know the type at the beginning only.
	`class  Test {`
	`public static void main(String[]  args) {`
	`Test  t = new Test( ) ;`
	`System.out.println(`
	`   Class.forName(args[0]).isInstance( ));`
	
	`//arg[0] --- We  don't know  the type`
	`                    at beginning`
	`  }`
	`}`
	
	`java Test Test    //true`
	`java Test String    //false`
	`java  Test Object    //true`

`int x= 10 ;`
`x=x++;`
`System.out.println(x);`
`               //10`
	1. consider old value of x for assignment x=10
	2. Increment x value x=11
	3. Perform assignment with old considered x value x=10