Operators

A symbol that tells the compiler or interpreter to perform specific mathematical, relational or logical operation.

Arithmetic operators

Arithmetic operators are the symbols that represent arithmetic math operations or an **arithmetic operator** is an **operator** that denotes that a specific **mathematical operation** is needed.

Increment and decrement operator

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

- We can apply increment or decrement operator only for variables but not for constant values.
- Nesting of increment or decrement operators is not allowed.

```
int y=++(++x);          //invalid
```

- We can't apply increment or decrement operator for final variables.

```
final int x=10
x++                     //invalid
```

- We can apply increment or decrement operator for any primitive type except Boolean.

```
char ch='a'             double d=10.5           boolean b=true
ch++                    d++                     b++
System.out.println(ch)  Syste.out.println(d)    Syste.out.println(b)
```

Output: b 11.5 CE

- If we apply any arithmetic operator between two variables "a" and "b" the result type is always.

```
max(int, typeof a, typeof b)
```

Example

```
byte a=10;
byte b=20;
byte c=a+b          //CE: possible loss of precision found: int required: byte
System.out.println(c);
```

Example

```
byte b=10;
b=b+1;//CE: possible loss of precision foud: int required: byte
System.out.println(c);
```

Solution is type casting (byte)(b+1)

- In the case of increment or decrement operator the required type-casting will be performed automatically by the compiler.

```
b++;    means
b=(type of b)(b+l);
b=(byte )(b+ 1);
```

Example:

```
byte b=10;
b++;
```

```
System.out.println(b);          //11
```

Arithmetic operators (+, -, *, /, %)

If we apply any arithmetic operation between two variables "a" and "b". The result type is always

max(int, type of a, type b)

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

Example

```
System.out.println('a'+l);              //98
System.out.println('a'+ 'b ') ;         //195
System.out.println(l0+0.5);             //10.5
System.out.println('a'+3.5);            //100.5
```

Infinity

In the case of integral arithmetic (byte, short, int, long) there is no way to represent infinity. Hence if infinity is the result then we will get ArithmeticException.

Example

```
System.out.println(l0/0);               //RE Exception in thread "main"
                                        java.lang.ArithmeticException: / by zero
```

But in floating point arithmetic (float, double), there is a way to represent infinity. For this Float and Double classes contains the following two constants.

POSITIVE_INFINITIVE;
NEGITIVE_INFINITIVE;

Therefore, if infinity is the result we won't get any runtime exception in floating point arithmetic.

Example

```
System.out.println( 10/0.0);        //+lnfinity
System.out.println(-10/0.0);        //-Infinity
```

NaN(Not a Number)

In the case of integral arithmetic there is no way to represent "undefined results". Hence if the result is undefined (0/0) we will get runtime exception saying

ArithmeticException.

Example

```
System.out.println(0/0);            //RE Exception In thread "main"
                                    java.lang.ArithmeticException:/ by zero
```

But in floating point arithmetic (float, double), there is a way to represent undefined result for this Float and Double classes contain "NaN". Therefore, if the result is undefined, we won't get any runtime exception in floating point arithmetic.

Example

```
System.out.println(0.0/0);          //NaN
System.out.println(-0.0/0);         //NaN
System.out.println(0/0.0);          //NaN
```

For any x value including NaN the following expressions return false.

Example

```
class Test
{
	public static void main(String[] args)
	{
		int x=10;
		System.out.println(x>Float.NaN);		//false
		System.out.println(x<Float.NaN);		//false
		System.out.println(x>=Float.NaN);		//false
		System.out.println(x<=Float.NaN);		//false
		System.out.println(x==Float. NaN );		//false
	}
}
```

For any x value including NaN the following expression return true.

Example

```
class Test
{
	public static void main{String[] args)
	{
		int x=10;
		System.out.println(x!=Float.NaN);		//true
		System.out.println(Float.NaN !=Float.NaN);	//true

	}
}
```

String concatenation operator

- The only operator which is overloaded in java is "+" operator. Sometime it acts as arithmetic addition operator and some time as concatenation operator.
- If at least one argument is string type then "+" operator acts as concatenation and if both arguments are number type then it acts as arithmetic addition operator.

Example

```
	String a =”CSE”;
	int b=10, c=20, d=30;
	System.out.println( a+b+c+d);		//CSE102030
	System.out.println(b+c+d+a);		//60CSE
	System.out.println(b+c+a+d);		//30CSE30
	System.out.println(b+a+c+d);		//10CSE2030
```

Example

```
	String a="CSE";
	int b=10, c=20, d=30;
	a=b+c+d;				//CE: incompatible types
	System.out.println( c);
```

Example

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

```
c=a+b+d;                          //CE: incompatible types
System.out.println(a);            //CSE1020
System.out.println(c);            //40
system.out.println(c);
```

Relational operator:(<,<=,>,>=)

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

Example

```
System.out.println( 10> 10.5 );          //false
System.out.println('a'>95.5);            //true
System.out.println('z'>'a');             //true
System.out.println(true>false);          //CE:operator > cannot be applied to boolean,boolean
```

We can't apply relational operators for the object types.

Example

```
System.out.println("cse">"cse");        //CE: operator > cannot be applied to
                                         java.lang.String,java.lang.String
```

We can't perform nesting of relational operators.

Example

```
System.out.println(10<20<30);            //CE operator< cannot be applied to boolean,int
        ↓                                System.out.println{10<20<30);
System.out.println(true<30);
```

Equality operator:(==, !=)

We can apply equality operators for every primitive type including boolean type also.

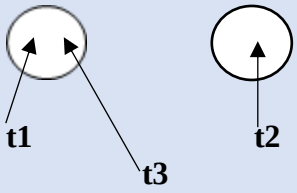
Example

```
System.out.println(10==10.0);            //true
System.out.println('a'==97.0);           //true
System.out. println(true==true );        //true
System.out.println('a' !='b');           //true
```

We can apply equality operator even for object reference also. In the case of object references == (double equal operator) is always meant for reference comparison only (address comparison). i.e. rl==r2 return true if and only if both rl and r2 point to the same object.

Example

```
Thread tl=new Thread ();
Thread t2=new Thread ();
Thread t3=tl;
System.out.println(tl==t2);              //false
System.out.println(tl==t3);              //true
```



To use equality operator compulsory there should be some relationship between argument type (either parent-child (or) child-parent (or) same type) otherwise we will get compile time error saying" incomparable types".

Example

```
Object o=new Object();
String s=new String("bhaskar");
StringBuffer sb=new StringBuffer();
System.out.println(o==s);                //false
System.out.println(o==sb );              //false
```

```
System.out.println(s==sb );                    //incomparable types: java.lang.String and
                                    java.lang.StringBuffer System.out.println(s==sb);
```

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

==Vs.equals()

==operator is always meant for reference comparison whereas .equals() method mostly meant for content comparison.

Example

```
String sl=new String("software");
String s2=new String("software");
System.out.println(sl==s2);              //false
System.out.println(s1.equals(s2));       //true
```

lnstanceof operator

We can use this operator to check whether the given object is of particular type (or) not.

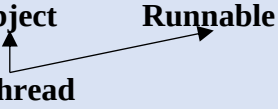
Syntax:

```
r instanceof x
```

where r is object reference and x is class/interface name

Example

```
Thread t=new Thread();
System.out.println(t instanceof Thread);      //true
System.out.println(t instanceof Object);      //true
System.out.println(t instanceof Runnable);    //true
```



To use "instanceof" operator compulsory there should be some relationship between argument types (either parent-child (or) child-parent (or) same type) otherwise we will get compile time error saying "inconvertible types".

Example

```
String s=new String("software");
        System.out.println(s instanceof Thread);        // inconvertible types
```

Example

```
Object o=new Object ();
System.out.println(o instanceof String);                //false
```

For any class or interface x '**null instanceof x'** the result is always "false".

Example

```
System.out.println(null instanceof String);             //false
```

Bitwise operators

& (AND)

If both arguments are true then 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.

Example

```
System.out.println(true & false);                       //false
System.out.println(true | false);                       //true
System.out.println(true ^ false);                       //true
```

- We can apply bitwise operators even for integral types (byte, short, int, long) also.

Example (operate according to size of data type i.e., on 32 bits)

```
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
```

Bitwise complement (~) (tilde symbol) operator:

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

Example

```
System.out.println(~true)    //CE: operator ~ cannot be applied to boolean

System.out.println(~4);      //5
```

int take 4 bytes that is we need to operate on all 32 bits for given integer number as:

4 00000……000000000100

11111……111111111011 (now msb is 1 that is number is negative and negative numbers are stored in memory in 2's complement form.

```
1111…….111111111011
0000…….000000000100 (1's complement)
                   1
0000……0000000000101 (2's complement)
```

Boolean complement (!) operator

- This operator is applicable only for boolean types but not for integral types.

```
System.out.println( !true);    //false
System.out.println( !false );  //true
```

Short circuit (&&, ||) operators

These operators are exactly same as normal bitwise operators &, I except the following differences.

&,|	&&,||
Both arguments should be evaluated always.	Second argument evaluation is optional.
Relatively performance is low.	Relatively performance is high.
Applicable for both integral and boolean types.	Applicable only for boolean types but not for integral types.

rl&&r2

r2 will be evaluated if and only if rl is true.

rl || r2

r2 will be evaluated if and only if rl is false.

Example

```
class OperatorsDemo
{
    public static void main(String[] args)
    {
        int x=10, y=15;
        if(++x> 10( operator )++y< 15)
```

```
		{
			++x;
		}
		else
		{
			++y;
		}
		System.out.println(x+"------------"+y);
	}
}
```

Output

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

```
class OperatorsDemo
{
	public static void main(String[] args)
	{
		int x=l0;
		if(++x<l0(operator)x/0>10)
		{
			System.out.println("hello");
		}
		else
		{
			System.out.println("hi");

		}
	}
}
```

&&	Output: Hi
&	Output: R.E: Exception in thread "main" java.lang.ArithmeticException: / by zero

Type-cast operator (primitive type casting)

There is two type of primitive type casting

Implicit type casting:

- Compiler is the responsible for this typecasting.
- Whenever we are assigning smaller data type value to the bigger data type variable this type casting will be performed.
- Also known as widening or up casting.
- There is no loss of information in this type casting.
- The following are various possible implicit type casting.

Example

```
	int x='a';
```

```
System.out.println(x);          //97
double d=10;
System.out.println(d);          //10.0
```

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.

Example

```
int x=130;
byte b=(byte)x;
System.out.println(b);          //-126
```

x(130)=000…..10000010 (int value in 32 bits)

byte b=(byte) x=10000010 (only 8 bits will be taken)
now MSB is 1 so number is positive and will be represented in 2s complement form

```
10000010
 1111101
       1
 1111110= -126
```

Whenever we are assigning bigger data type value to the smaller data type variable by explicit type casting the most significant bit(MSB)will be lost.

Example

```
int x=l50;
short s=(short)x;
System.out.println(s);          //150
byte b=(byte)x;
System.out.println(b);          //-106
```

Whenever we are assigning floating point data types to the integer data type by explicit type casting the digits after the decimal point will be loosed.

Example:

```
float x=l50.1234f;
double d=130.456;
int i=(int)x;
int j=(int)d;
System.out.println(i);          //150
System.out.println(j);          //130
```

Assignment operators

They are three types of assignment operators.

Simple assignment
Example

```
int x=10;
```

Chained assignment
Example

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

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

```
Int a=b=c=d=20;
```

Compound assignment
Sometimes we can mix assignment operator with some other operator to form compound assignment operator. The following is the list of all possible compound assignment operators in java.

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

Example

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

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

Conditional operator
The only ternary operator which is available in java is conditional operator.
Example

```
int x=(10>20)?30:40;
System.out.printl n(x) ;                    // 40
```

We can perform nesting of conditional operator also.
Example

```
int x=(10>20)?30:((100>20)?40:50);
System.out.prmtln(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<2)?30:40;
System.out.println(cl);                     //40
System.out.println(c2);                     //30
```

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

[] operator
We can use this operator to declare and construct arrays.

Java operator precedence
Unary operators
[], x++, X--,
++x, --x, ~, !
new, <type>

Arithmetic operators
*,/,%
+,-

Shift operators
>>,>>>,<<

Compression operators
<, <=,>,>=, instanceof

Equality operators
==,!=

Bitwise operators
&, ^, |

Short circuit operators
&&, ||

Conditional operator
(?:)

Assignment operators
+= I -=*=I=%=
Example
```
int i=l;
i+=++i + i++ + ++i + i++;
System.out.println(i);                    //13
```

Analysis
```
i=i+ ++i + i++ + ++i + i++;
i=1+2+2+4+4;
i=13;
```