Working with Operators and the Math API

Esteban Herrera

Author

@eh3rrera | eherrera.net

Operator

Symbol that tells the compiler to perform specific mathematical or logical operations.

Operator Types

Operator Type	Operators	Description

Operator precedence determines which operations are evaluated first in an expression.

Operator Precedence

Highest

Lowest

Category	Operators	Associativity
Postfix	expr++ expr--	Left to right
Unary	++expr --expr +expr -expr ~ !	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >> >>>	Left to right
Relational	< > <= >= instanceof	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Ternary	? :	Right to left
Assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=	Right to left

Left-to-right Evaluation

a + b − c

1st 2nd

Right-to-left Evaluation

a = b = c = 5

3rd 2nd 1st

Operator Precedence

Highest

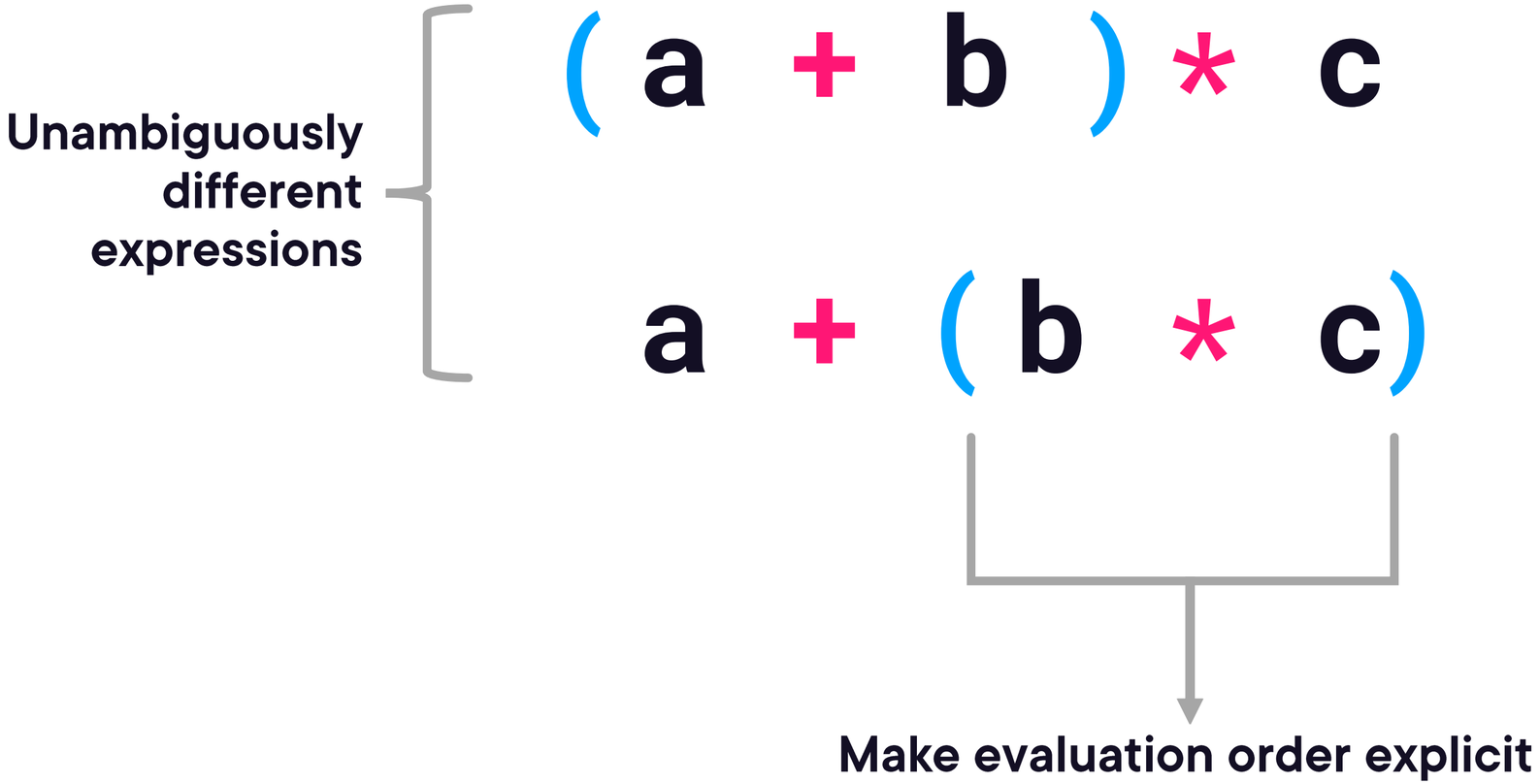
Lowest

Category	Operators	Associativity
Postfix	expr++ expr--	Left to right
Unary	++expr --expr +expr -expr ~ !	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >> >>>	Left to right
Relational	< > <= >= instanceof	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Ternary	? :	Right to left
Assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=	Right to left

Important Precedence Rules

- Postfix operations like x++ happen before prefix ones like ++x
- Multiplicative operations (*, /, %) happen before additive ones (+, -)
- Bitwise operations (&, |, ^) happen after comparisons (>, ==, etc.) but before logical ones (&&, ||)
- Assignments evaluate last, and right-to-left

Parentheses Usage



Parentheses Usage

```
(((a + (b * c)) - ((a / c) + b)) * c)
```

```
(a + b * c - (a / c + b)) * c
```

Using Unary Operators

Negation Operator

$$x = 5$$

$$-x = -5$$

Unary Complement Operator

Also called bitwise complement

Inverts all bits in a number

- `0000` → `1111`
- `1111` → `0000`

Java represents integers using 32 bits in two's complement format

Two’s Complement Format

+5 → 0000 0000 0000 0000 0000 0000 0000 0101

Two's Complement Format

-6 → `1111 1111 1111 1111 1111 1111 1111 1010`

1. Invert the bits

Two's Complement Format

-5 → `1111 1111 1111 1111 1111 1111 1111 1011`

2. Add 1

Leftmost Bit Serves as the Sign Bit

Bitwise Complement Operator

+5 → 0000 0000 0000 0000 0000 0000 0000 0101

-6 → 1111 1111 1111 1111 1111 1111 1111 1011

Bitwise Complement Operator

```
~5 = -6
```

```
~n = -(n + 1)
```

Bitwise Complement Operator

-5 → 1111 1111 1111 1111 1111 1111 1111 1011

+4 → 0000 0000 0000 0000 0000 0000 0000 0100

~(-5) = +4

Bitwise Complement Operator

Positive n

$$\sim n = -(n + 1)$$

Negative n

$$\sim(-n) = |n| - 1$$

Increment and Decrement Operators

```
int x = 5, y = 5;

int a = ++x;  // Prefix increment: x becomes 6, then a gets 6
int b = --x;  // Prefix decrement: x becomes 5, then b gets 5

int c = y++;  // Postfix increment: c gets 5, then y becomes 6
int d = y--;  // Postfix decrement: d gets 6, then y becomes 5
```

Increment and Decrement Operators

Operator	Name	Description
++x	Prefix increment operator	Increments x by 1, then returns the new value of x
x++	Postfix increment operator	Returns the current value of x, then increments x by 1
--x	Prefix decrement operator	Decrements x by 1, then returns the new value of x
x--	Postfix decrement operator	Returns the current value of x, then decrements x by 1

Increment and Decrement Operators

```
int x = 6, y = 6;

++x; // Same effect as x++
x++; // Same effect as ++x

int expression = y++ * ++y - 7; // ?
```

Using Binary (Arithmetic) Operators

Binary Arithmetic Operators

Addition
+

Subtraction
-

Multiplication
*****

Division
/

Modulus
%

Using Bitwise and Shift Operators

Bitwise Operators

AND
&

OR
|

XOR
^

Complement
~

Bitwise AND Operator

```
int a = 0b1010; // 10
int b = 0b1100; // 12

a & b = 0b1000  // 8
```

Bitwise OR Operator

```
int a = 0b1010; // 10
int b = 0b1100; // 12

a | b = 0b1110  // 14
```

Bitwise Exclusive OR Operator

```
int a = 0b1010; // 10
int b = 0b1100; // 12

a ^ b = 0b0110  // 6
```

Shift Operators

Left Shift
<<

Signed Right Shift
>>

Unsigned Right Shift
>>>

Left Shift Operator

```
int a = 0b1010; // 10
a << 1 = 0b10100 // 20

0000 1010  // Binary for 10
0001 0100  // Binary for 20

a << x = a × 2^x
10 << 1 = 10 × 2^1
        = 10 × 2
        = 20
```

Signed Right Shift Operator

```
int a = 0b1010; // 10
a >> 1 = 0b0101  //  5

0000 1010  // Binary for 10
0000 0101  // Binary for  5

a >> x = a / 2^x
10 >> 1 = 10 / 2^1
        = 10 / 2
        = 5
```

Unsigned Right Shift Operator

```
int a    = 0b11111111111111111111111111110110; // -10
a >>> 1 = 0b01111111111111111111111111111011 // 2147483643

// Before shift (-10 as binary)
11111111111111111111111111110110

// After >>> 1 (Shift right by 1, fill left with 0)
01111111111111111111111111111011
```

Using Assignment Operators

Compound Assignment Operators

Operator	Description	Equivalent Expression
+=	Addition assignment	a += b → a = a + b
-=	Subtraction assignment	a -= b → a = a - b
*=	Multiplication assignment	a *= b → a = a * b
/=	Division assignment	a /= b → a = a / b
%=	Modulus assignment	a %= b → a = a % b
&=	Bitwise AND assignment	a &= b → a = a & b
|=	Bitwise OR assignment	a |= b → a = a | b
^=	Bitwise XOR assignment	a ^= b → a = a ^ b
<<=	Left shift assignment	a <<= b → a = a << b
>>=	Signed right shift assignment	a >>= b → a = a >> b
>>>=	Unsigned right shift assignment	a >>>= b → a = a >>> b

Compound Assignment Operators

```
variable op= expression;
```

```
variable = variable op expression;
```

Compound Assignment Operators

```
int a = 10;

a += 5;  // equivalent to a = a + 5; a is now 15
a -= 3;  // equivalent to a = a - 3; a is now 12
a *= 2;  // equivalent to a = a * 2; a is now 24
a /= 4;  // equivalent to a = a / 4; a is now 6
a %= 5;  // equivalent to a = a % 5; a is now 1

int b = 0b1010; // binary representation of 10

b &= 0b1100;  // equivalent to b = b & 0b1100; b is now 0b1000 (8 in decimal)
b |= 0b0101;  // equivalent to b = b | 0b0101; b is now 0b1101 (13 in decimal)
b ^= 0b1001;  // equivalent to b = b ^ 0b1001; b is now 0b0100 (4 in decimal)
b <<= 2;      // equivalent to b = b << 2; b is now 0b10000 (16 in decimal)
b >>= 1;      // equivalent to b = b >> 1; b is now 0b01000 (8 in decimal)
b >>>= 2;     // equivalent to b = b >>> 2; b is now 0b00010 (2 in decimal)
```

Using Relational Operators

Relational Operators

Less Than
<

Greater Than
>

Less Than or Equal To
<=

Greater Than or Equal To
>=

The instanceOf Operator

```
object_reference instanceof Type
```

Pattern Matching with the instanceOf Operator

```
if (object_reference instanceof Type variable) {
    // Inside this block, variable is
    // automatically cast to Type
}
```

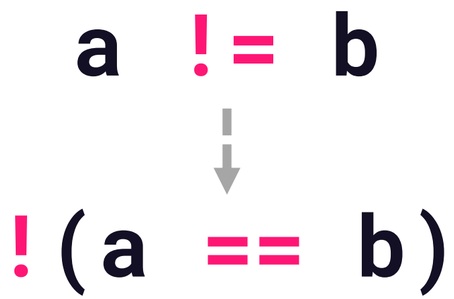
Using Equality Operators

Equality Operators

Equal To
==

Not Equal To
!=

Not Equal To



Rules for Overriding the Equals Method

Symmetry: If a.equals(b) is true, b.equals(a) must be true

Reflexivity: a.equals(a) must be true

Transitivity: If a.equals(b) and b.equals(c) are true, a.equals(c) must be also true

Consistency: The result must be the same if objects aren't modified

Non-nullity: a.equals(null) must return false

Using Logical Operators

Logical Operators

Logical AND
&

Logical Or
|

Logical XOR
^

Short-circuit
Logical AND
&&

Short-circuit
Logical OR
||

Logical NOT
!

Truth Tables

Logical AND		
a	b	a & b
false	false	false
false	true	false
true	false	false
true	true	true

AND (&) is true only if both operands are true.

Logical OR		
a	b	a | b
false	false	false
false	true	true
true	false	true
true	true	true

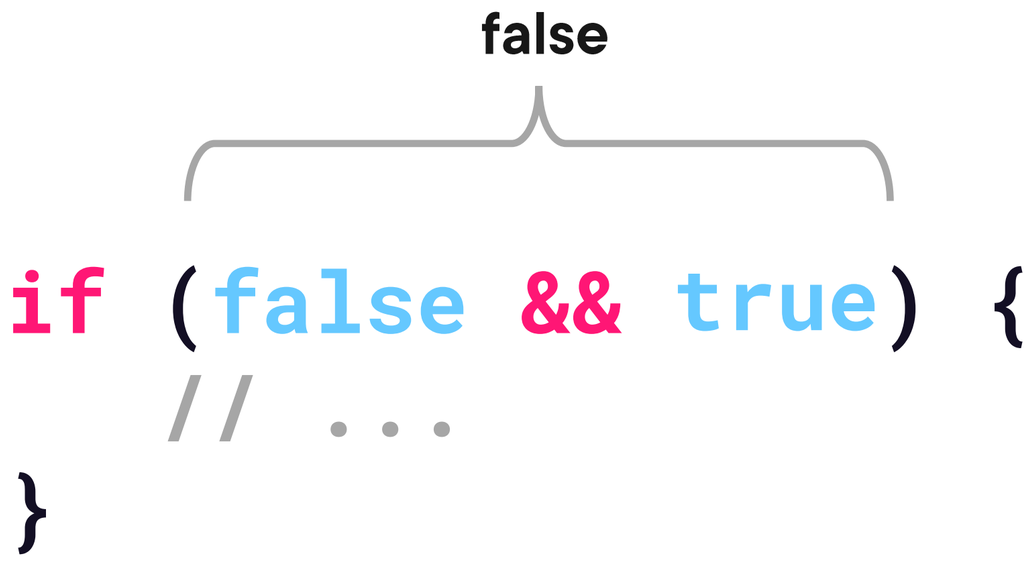
OR (|) is true if at least one operand is true.

Logical XOR		
a	b	a ^ b
false	false	false
false	true	true
true	false	true
true	true	false

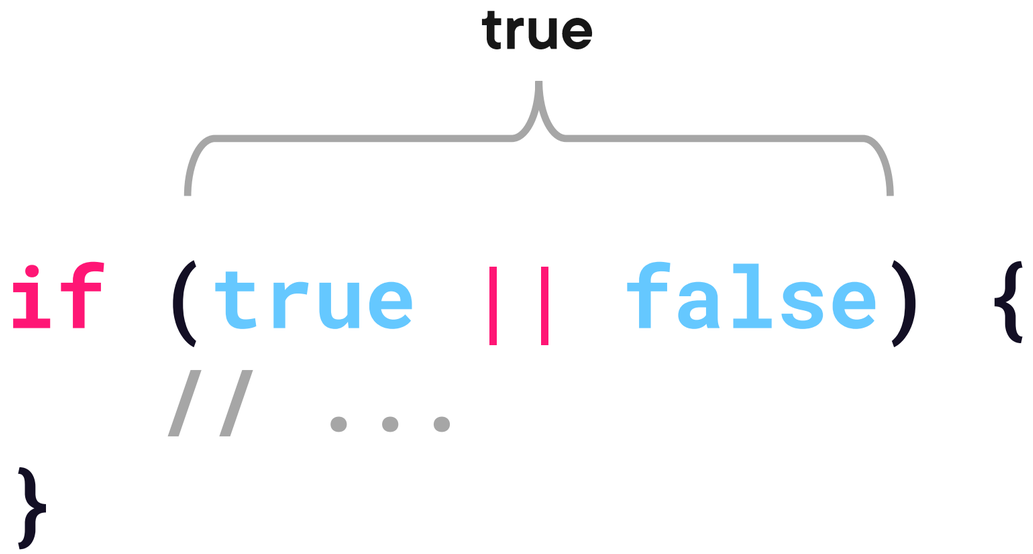
XOR (^) is true if exactly one operand is true.

Short-circuit operators (&& and ||) only evaluate the second operand when necessary.

Short-circuit AND



Short-circuit OR



Avoiding a NullPointerException

```
String str = null;
if (str != null && str.length() > 0) {
    // This code will not throw a NullPointerException
}
```

Be Careful with Side Effects

```
int a = 10;
if (a > 5 && ++a > 10) {
    // The value of a will be 11 if a > 5, but it will
    // remain 10 if a <= 5 because, in this case,
    // ++a won't be executed
}
```

Promotion Rules

Numeric Promotion

Promotion Rules

If one operand is double, the other becomes double

Otherwise, if one operand is float, the other becomes float

Otherwise, if one operand is long, the other becomes long

Otherwise, both operands become int

Promotion Rules

```
int a = 10;
double b = 20.0;
double result1 = a + b; // a is promoted to double

float c = 10.0F;
long d = 20L;
float result2 = c + d; // d is promoted to float

short e = 10;
short f = 20;
int result3 = e + f; // e and f are promoted to int
```

Promotion Rules

```
// Unary operators
byte b = 5;
int result = -b; // b is promoted to int before applying
                 // the unary minus operator

// Compound assignment operators
int i = 10;
long l = 5L;
i += l; // Equivalent to i = (int)(i + l);
```

The Math API

```
public static double min(double a, double b)
public static float min(float a, float b)
public static int min(int a, int b)
public static long min(long a, long b)

public static double max(double a, double b)
public static float max(float a, float b)
public static int max(int a, int b)
public static long max(long a, long b)
```

Math.min and Math.max Methods

Find the minimum and maximum of two values

```
public static double pow(double a, double b)
```

Math.pow Method

Returns the first argument raised to the power of the second argument

Math.pow Rules

Any number raised to 0 returns 1.0

Any number raised to 1 returns itself

If either argument is NaN (Not-a-Number), the result is NaN, except when the base is 1, which returns 1.0

For negative bases:

- Odd integer exponents keep the sign of the base. Even integer exponents behave as if the base were positive
- If the exponent is not an integer, the result is NaN

```
// These methods return the closest int or long to the argument. Halfway values
// (like 0.5) are rounded up, following the round half up convention
public static int round(float f)
public static long round(double d)
// Returns the double value that is closest in value to the argument and is equal to a
// mathematical integer. If two double values are equally close, the even one is chosen
public static double rint(double d)
// Returns the largest (closest to positive infinity) double value that is less than or
// equal to the argument and is equal to a mathematical integer
public static double floor(double d)
// Returns the smallest (closest to negative infinity) double value that is greater than
// or equal to the argument and is equal to a mathematical integer
public static double ceil(double d)
```

Rounding Methods

```
public static double random()
```

Math.random Method

Returns positive double value greater than or equal to 0.0 and less than 1.0

Review

Operators Overview

Operator Type	Operators	Description
Unary	+ - ++ -- ! ~	Operate on a single operand
Arithmetic	+ - * / %	Perform basic mathematical operations
Relational	> < >= <= == !=	Compare values and return boolean results
Bitwise	& | ^ ~ << >> >>>	Work at the bit level
Logical	& | && || !	Used for boolean logic
Assignment	= += -= *= /= %= &= |= ^= <<= >>= >>>=	Assign values with optional operations
Ternary	condition ? value_if_true : value_if_false	Short-form conditional selection

Operator Precedence

Highest

Lowest

Category	Operators	Associativity
Postfix	expr++ expr--	Left to right
Unary	++expr --expr +expr -expr ~ !	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >> >>>	Left to right
Relational	< > <= >= instanceof	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Ternary	? :	Right to left
Assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=	Right to left

Practice Questions

Test your knowledge.

Question 1

What will be the output of the following program?

```java
public class OperatorPrecedenceTest {
    public static void main(String[] args) {
        int a = 2, b = 3, c = 4;
        int result = a + b * c / b - c % a;
        System.out.println(result);
    }
}
```

```
2 + 3 * 4 / 3 - 4 % 2
2 + 12 / 3 - 4 % 2
2 + 4 - 4 % 2
2 + 4 - 0
6
```

A) 4 

B) 6 

C) 5 

D) 3 

Question 2

What will be the output of the following program?

```
public class MathTest {
    public static void main(String[] args) {
        double a = -5.5, b = 3.2;
        int c = 2;
        double result =
           Math.ceil(a) + Math.floor(b) + Math.pow(c, c) + Math.min(a, b) + Math.round(b);
        System.out.println(result);
    }
}
```

```
Math.ceil(-5.5) + Math.floor(3.2) + Math.pow(2, 2) + Math.min(-5.5, 3.2) + Math.round(3.2)
-5.0 + Math.floor(3.2) + Math.pow(2, 2) + Math.min(-5.5, 3.2) + Math.round(3.2)
-5.0 + 3.0 + Math.pow(2, 2) + Math.min(-5.5, 3.2) + Math.round(3.2)
-5.0 + 3.0 + 4.0 + Math.min(-5.5, 3.2) + Math.round(3.2)
-5.0 + 3.0 + 4.0 + (-5.5) + Math.round(3.2)
-5.0 + 3.0 + 4.0 + (-5.5) + 3.0
-0.5
```

A) -0.5 ✓

B) 3.0 ✗

C) 5.0 ✗

D) Compilation error ✗