# Supplement: How Operators and Operands are Evaluated in Java
## For Introduction to Java Programming
## By Y. Daniel Liang

The text introduces operator precedence and associativity. This supplement goes into more details on how operators and operands are evaluated in Java.

## 1 Operator Precedence and Associativity

Operator precedence and associativity determine the order in which operators are evaluated. Suppose that you have this expression:

```
3 + 4 * 4 > 5 * (4 + 3) - 1
```

What is its value? What is the execution order of the operators? Arithmetically, the expression in the parentheses is evaluated first. (Parentheses can be nested, in which case the expression in the inner parentheses is executed first.) When evaluating an expression without parentheses, the operators are applied according to the precedence rule and the associativity rule. The precedence rule defines precedence for operators, as shown in Table 1, which contains the operators you have learned so far. Operators are listed in decreasing order of precedence from top to bottom. Operators with the same precedence appear in the same group. (See Appendix C, "Operator Precedence Chart," for a complete list of Java operators and their precedence.)

**Table 1**
*Operator Precedence Chart*

| Precedence | Operator |
|---|---|
| ↓ | **var++** and **var--** (Postfix) |
| | **+**, **-** (Unary plus and minus), **++var** and **--var** (Prefix) |
| | (type) (Casting) |
| | **!** (Not) |
| | *****, **/**, **%** (Multiplication, division, and remainder) |
| | **+**, **-** (Binary addition and subtraction) |
| | **<**, **<=**, **>**, **>=** (Comparison) |
| | **==**, **!=** (Equality) |
| | **^** (Exclusive OR) |
| | **&&** (AND) |
| | **\|\|** (OR) |
| | =, +=, -=, *=, /=, %= (Assignment operator) |

If operators with the same precedence are next to each other, their *associativity* determines the order of evaluation. All binary operators except assignment operators are *left-associative*. For example, since + and – are of the same precedence and are left-associative, the expression

```
a - b + c - d  ═══ equivalent ═══  ((a - b) + c) - d
```

Assignment operators are *right-associative*. Therefore, the expression

```
a = b += c = 5   ═══ equivalent ═══   a = (b += (c = 5))
```

Suppose **a**, **b**, and **c** are 1 before the assignment; after the whole expression is evaluated, **a** becomes 6, **b** becomes 6, and **c** becomes 5. Note that left associativity for the assignment operator would not make sense.

Arithmetically, the expression **3 + 4 * 4 > 5 * (4 + 3) - 1** is evaluated as shown in Figure 1a. This evaluation is known as the *arithmetic evaluation*.

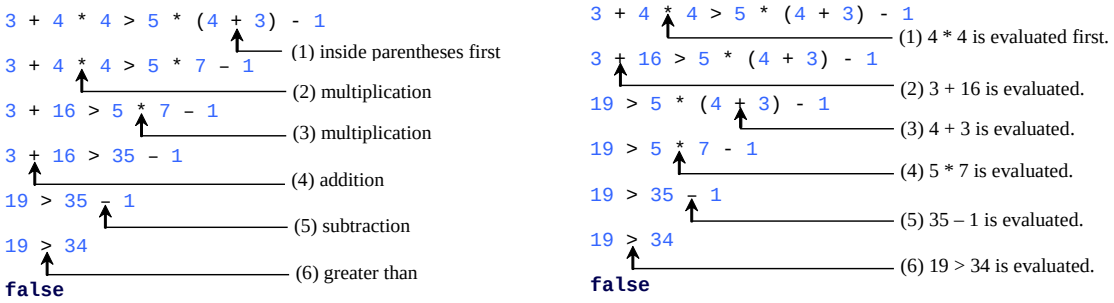


**Figure 1**
*(a) The expression is evaluated arithmetically. (b) The expression is evaluated in Java.*

However, in Java, the operators are evaluated from left to right as long as it does not violate the precedence and associativity. This is known as the *Java expression evaluation* in contrast to the arithmetical evaluation. In Java, the expression **3 + 4 * 4 > 5 * (4 + 3) - 1** in Figure 3.5(a) is evaluated as shown in Figure 3.5(b). Java expression evaluation is equivalent to the arithmetic evaluation. Java expression evaluation is easier to implement and more efficient to execute.

Here are two more examples. In the expression **1 + 2 + 3 * 4**, **1 + 2** is evaluated before **3 * 4**, as shown in Figure 2a. In the expression **1 + 3 * 4**, **3 * 4** is evaluated first, and then **1 + 12** is evaluated, as shown in Figure 2b.

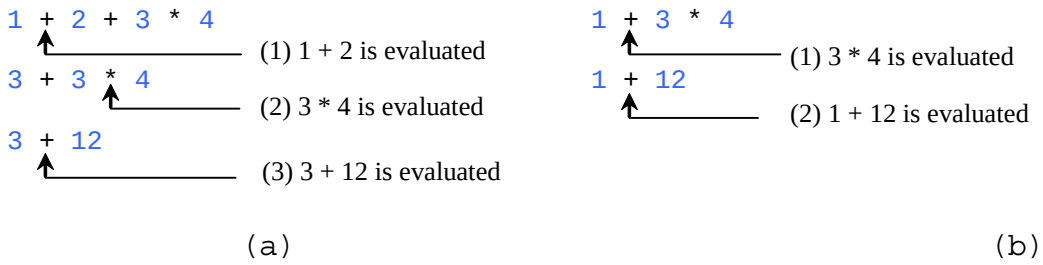


**Figure 2**
*The expression is evaluated in Java.*

> TIP
> You can use parentheses to force an evaluation order as well as to make a program easy to read. Use of redundant parentheses does not slow down the execution of the Java expression.

## 2 Operand Evaluation Order

The precedence and associativity rules specify the order of the operators but not the order in which the operands of a binary operator are evaluated. Operands are evaluated strictly from left to right in Java. *The left-hand operand of a binary operator is evaluated before any part of the right-hand operand is evaluated.* This rule takes precedence over any other rules that govern expressions. Consider this expression:

```
a + b * (c + 10 * d) / e
```

**a**, **b**, **c**, **d**, and **e** are evaluated in this order. *If no operands have side effects that change the value of a variable, the order of operand evaluation is irrelevant.* Interesting cases arise when operands do have a side effect. For example, **x** becomes **1** in the following code because **a** is evaluated to **0** before **++a** is evaluated to **1**.

```
int a = 0;
int x = a + (++a);
```

But **x** becomes **2** in the following code because **++a** is evaluated to **1**, then **a** is evaluated to **1**.

```
int a = 0;
int x = ++a + a;
```

*The order for evaluating operands takes precedence over the operator precedence rule.* In the former case, **(++a)** has

higher precedence than addition (**+**), but since **a** is a left-hand operand of the addition (**+**), it is evaluated before any part of its right-hand operand (e.g., **++a** in this case).

> NOTE:
>
> The order of evaluating operands does not matter if the expression does not have the **++** and **--** operators. The **++** and **--** operators can have side effects that change the value of a variable in an expression. So, to avoid errors, do not use these operators in expressions that modify multiple variables or the same variable multiple times.