

Precedence of Operators

This is a good time to re-visit a topic discussed earlier, operator precedence. Table 1 lists all the operators in Java. Most of the operators in the table have been discussed previously. The others will be presented later on an as-needed basis. Each row represents a different precedence level, with operators farther up the table having higher precedence than operators farther down.

Table 1. Precedence of Operators (complete)

Precedence	Operator(s)	Operation	
highest	[] . () expr++ expr--	postfix	
	++expr --expr +expr -expr	unary	
	new (type)expr	creation or cast	
	* / %	multiplicative	
	+ -	additive	
	<< >> >>>	shift	
	> < >= <= instanceof	relational	
	== !=	equality	
	&	bitwise AND	
	^	bitwise exclusive OR	
		bitwise inclusive OR	
	&&	logical AND	
		logical OR	
	?:	conditional	
	lowest	= op=	assignment

All binary operators — those receiving two arguments — are *left-associative*, meaning they are executed left-to-right. In other words,

$$4 + 5 - 6 + 7$$

is the same as

$$((4 + 5) - 6) + 7$$

Of course, if binary operators from different rows in Table 1 are mixed in an expression, then the position in the table determines the order of evaluation. So,

$$5 - 6 * 3$$

is the same as

$$5 - (6 * 3)$$

Note that the logical AND operator (&&) is of higher precedence than the logical OR (||) operator. So,

$$a \&\& b \ || \ c \ \&\& \ d$$

is the same as

```
(a && b) || (c && d)
```

Although parentheses can always be added for clarity, or “just to make sure”, try to avoid excessive use. Often the result is *less* clarity. It’s a good idea to gain familiarity with operator precedence, and to use parentheses sparingly — only when necessary to override the natural precedence of operators.

The assignment operator (=) is *right-associative*. So,

```
a = b = c
```

is the same as

```
a = (b = c)
```

It is common in Java to mix assignment with a boolean test, for example

```
String s;  
if ((s = stdin.readLine()) != null)  
    /* process line */
```

Since the assignment operator (=) is of lower precedence than the inequality operator (!=), an extra set of parentheses is needed. Reworking the above fragment as

```
String s;  
if (s = stdin.readLine() != null)    // WRONG!  
    /* process line */
```

results in a compiler error.

The op= entry along the bottom row in Table 1 implies any of

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

Bear in mind that operator precedence combined with associativity determines the order of evaluation. So, with an expression such as

```
a + b + c
```

the compiler first evaluates *a*, then evaluates *b*, then adds the values of *a* and *b*, then evaluates *c*, then adds the value of *c* to the previous result. Order of evaluation matters, in particular, if there are side effects of any kind. Consider the following code fragment:

```
int a = 1;  
int b = 2;  
int c = 3;  
int d = a + b++ + c + b;  
System.out.println("d = " + d);
```

Can you determine the output? We’ll leave it for you to explore this.