

is equivalent to

```
x--;
```

These operators are unique in that they can appear both in *postfix* form, where they follow the operand as just shown, and *prefix* form, where they precede the operand. In the foregoing examples, there is no difference between the prefix and postfix forms. However, when the increment and/or decrement operators are part of a larger expression, then a subtle, yet powerful, difference between these two forms appears. In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression. In postfix form, the previous value is obtained for use in the expression, and then the operand is modified. For example:

```
x = 42;  
y = ++x;
```

In this case, **y** is set to 43 as you would expect, because the increment occurs *before* **x** is assigned to **y**. Thus, the line **y = ++x;** is the equivalent of these two statements:

```
x = x + 1;  
y = x;
```

However, when written like this,

```
x = 42;  
y = x++;
```

the value of **x** is obtained before the increment operator is executed, so the value of **y** is 42. Of course, in both cases **x** is set to 43. Here, the line **y = x++;** is the equivalent of these two statements:

```
y = x;  
x = x + 1;
```

The following program demonstrates the increment operator.

```
// Demonstrate ++.  
class IncDec {  
    public static void main(String args[]) {  
        int a = 1;  
        int b = 2;  
        int c;  
        int d;  
        c = ++b;  
        d = a++;  
        c++;  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
        System.out.println("c = " + c);  
        System.out.println("d = " + d);  
    }  
}
```


Operator Precedence

Table 4-1 shows the order of precedence for Java operators, from highest to lowest. Operators in the same row are equal in precedence. In binary operations, the order of evaluation is left to right (except for assignment, which evaluates right to left). Although they are technically separators, the `[]`, `()`, and `.` can also act like operators. In that capacity, they would have the highest precedence. Also, notice the arrow operator `(->)`. It is used in lambda expressions.

Using Parentheses

Parentheses raise the precedence of the operations that are inside them. This is often necessary to obtain the result you desire. For example, consider the following expression:

```
a >> b + 3
```

This expression first adds 3 to **b** and then shifts **a** right by that result. That is, this expression can be rewritten using redundant parentheses like this:

```
a >> (b + 3)
```

However, if you want to first shift **a** right by **b** positions and then add 3 to that result, you will need to parenthesize the expression like this:

```
(a >> b) + 3
```

Highest						
++ (postfix)	-- (postfix)					
++ (prefix)	-- (prefix)	~	!	+ (unary)	- (unary)	(type-cast)
*	/	%				
+	-					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
&&						
?:						
->						
=	op=					
Lowest						

Table 4-1 The Precedence of the Java Operators

However, if you want to first shift **a** right by **b** positions and then add 3 to that result, you will need to parenthesize the expression like this:

`(a >> b) + 3`

Highest						
++ (postfix)	-- (postfix)					
++ (prefix)	-- (prefix)	~	!	+ (unary)	- (unary)	(type-cast)
*	/	%				
+	-					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
&&						
?:						
->						
=	op=					
Lowest						

Table 4-1 The Precedence of the Java Operators