

## Operators

Representing and storing primitive data types is, of course, essential for any computer language. But, so, too, is the ability to perform operations on data. Java supports a comprehensive set of *operators* to facilitate adding, subtracting, multiplying, dividing, comparing, etc. Let's start with a simple demonstration program. Figure 1 contains the listing for a program that adds one and one to get two.

```
1 public class Numbers
2 {
3     public static void main(String[] args)
4     {
5         int x;
6         x = 1;
7         int y = 1;
8         int z = x + y;
9         System.out.print("1 + 1 = ");
10        System.out.println(z);
11    }
12 }
```

Figure 1. Numbers.java

The overall structure of this program is identical to the Java applications we saw in the previous chapter. Now, however, we are operating on data. When this program executes, it generates the following output:

```
1 + 1 = 2
```

This is pretty trivial, but don't be fooled. There are many aspects of the Java language sitting before us in Figure 1. Let's walk through the program's use of data types and operators.

In line 5, an `int` variable named `x` is declared. The effect is to set aside storage for `x`, but the storage is not initialized with a value (see Figure 2a). In line 6, the variable `x` is assigned the value 1. The equals sign (`=`) is known as the *assignment* operator. The effect is to place the value 1 into the storage location set aside for the variable `x` (see Figure 2b).



Figure 2. (a) A variable `x` is declared, but not initialized  
(b) a variable `x` is initialized with the value 1.

In line 7, we see declaration combined with assignment. A new `int` variable, `y`, is declared and initialized with the value 1.

Line 8 contains another example of declaration combined with assignment. An `int` variable `z` is declared and assigned the value `x + y`. The plus sign (`+`) is the *addition* operator. Since line 8 contains two operators, there is a need to process the operations in a certain order. This process is governed by Java's *precedence of operators*. As you might guess, addition takes precedence over assignment; so, the expression `x + y` is evaluated first and then the result is assigned to `z`. The result is printed in lines 9 and 10.

Java also includes operators for *subtraction* (-), *multiplication* (\*), and *division* (/). Where these and other operators are mixed, precedence must be carefully considered. Let's examine precedence of operators in more detail. The program `Operators.java` (Figure 3) shows subtraction and multiplication combined in a single expression.

```
1 public class Operators
2 {
3     public static void main(String[] args)
4     {
5         int i = 12 - 5 * 3;
6         int j = (12 - 5) * 3;
7
8         System.out.println(i);
9         System.out.println(j);
10    }
11 }
```

Figure 3. `Operators.java`

The output of this program is

```
-3
21
```

In line 5, an `int` variable `i` is declared and assigned the result of the expression `12 - 5 * 3`. If we evaluate the expression left-to-right the answer is 21, which is wrong because multiplication takes precedence over subtraction. The expression `5 * 3` is evaluated first and the result, 15, is then subtracted from 12 to yield the correct answer of -3. To perform the subtraction first, if desired, parentheses are added as shown in line 6. Parentheses override the natural precedence of operators.

Division and multiplication are of equal precedence, as are addition and subtraction. When arithmetic operators of the same precedence appear together, they are evaluated left to right. So, the expression `2 + 3 - 5 - 6` equals -6 and the expression `30 * 4 / 3 / 8` equals 5.

### ***The Remainder Operator***

When dividing integers, the decimal portion of the result, or the remainder, if any, is lost. So, the expression `13 / 5` equals 2. The remainder after division of integers can be obtained using the *remainder* (%) operator. (The remainder operator is sometime called the *modulus* or *mod* operator). So, the expression `13 % 5` equals 3, because 13 divided by 5 is 2 with a remainder of 3. The program `Days` (Figure 4) demonstrates the mod operator in action.

```

1 public class Days
2 {
3     public static void main(String[] args)
4     {
5         int seconds = 1000000; //one million
6
7         int days      = seconds / (24 * 60 * 60);
8         int remainder = seconds % (24 * 60 * 60);
9         int hours     = remainder / (60 * 60);
10        remainder    = remainder % (60 * 60);
11        int minutes   = remainder / 60;
12        remainder    = remainder % 60;
13
14        System.out.println(seconds + " seconds = ");
15        System.out.println(days   + " days");
16        System.out.println(hours  + " hours");
17        System.out.println(minutes + " minutes, and");
18        System.out.println(remainder + " seconds");
19    }
20 }

```

Figure 4. Days . java

The Days . java program computes the number of days, hours, minutes, and seconds in one million seconds. The output is

```

1000000 seconds =
11 days
13 hours
46 minutes, and
40 seconds

```

In line 7, the number of days in one million seconds is computed by dividing the `int` variable `seconds` by  $(24 * 60 * 60)$ . The result is 11. The arithmetic is for integers, so the remainder is lost. The remainder is retrieved in line 8 by performing the same operation again, except using the mod operator. The result — the remainder — is assigned to the `int` variable `remainder`. The number of hours is computed by dividing `remainder` by  $(60 * 60)$  in line 9. A similar process is repeated for hours, minutes, and seconds (lines 10-12).

When one or both values are negative, extra caution is warranted. The operations must obey the following rule:

$$(x / y) * y + x \% y == x$$

So, for example

```

7 % 2      equals 1
7 % -2     equals 1
-7 % 2     equals -1, and
-7 % -2    equals -1.

```

Although much less used, the remainder operator also works with floating-point numbers. In this case, it returns the remainder after "even" division, for example, the expression  $8.0 \% 2.5$  returns 0.5.

The `%` operator has the same precedence as multiplication and division.

## ***Precedence of Operators***

Although Java includes many more operators, let's summarize what we have learned thus far about the precedence of operators (see *Table 1*).

*Table 1. Precedence of Operators*

| Precedence   | Operator(s) | Operation                           | Association    |
|--------------|-------------|-------------------------------------|----------------|
| highest      | ( )         | override natural precedence         | Inner to outer |
| next highest | * / %       | multiplication, division, remainder | L to R         |
| next highest | + -         | addition, subtraction               | L to R         |
| lowest       | =           | assignment                          | R to L         |

The association is left-to-right for most operators. One exception is the assignment operator. In the event of more than one assignment operator in a single statement, the association is right to left. Consider, for example, the following statements:

```
int x = 2;
int y = 3;
int z;
int a = z = x + y;
```

In final line, the expression `x + y` is evaluated first, then the result is assigned to `z`. Finally, the value of `z` is assigned to `a`.

## ***Promotion of int to double***

It is often necessary to combine integer and floating-point numbers in the same expression. Java permits this; however, the following two points must be considered:

- Operations between an `int` and an `int` always yeild an `int` (the remainder is lost).
- When an `int` and a `double` are mixed in an expression, the `int` is "promoted" to a `double` and the result is a `double`.

*Promotion* is the act of automatically converting a "lower" type to a "higher" type. The program `Hours.java` illustrates promotion by mixing `int` and `double` variables in expressions (see Figure 5).

```

1 public class Hours
2 {
3     public static void main(String[] args)
4     {
5         int seconds = 1000000; // one million
6         int factorInt = 3600;
7         double factorDouble = 3600.0;
8
9         double hours = seconds / factorInt;
10        System.out.println(hours);
11
12        hours = seconds / factorDouble;
13        System.out.println(hours);
14    }
15 }

```

Figure 5. Hours.java

This program computes the number of hours in one million seconds, and generates the following output:

```

277.0
277.77777777777777

```

The result is computed twice using the same values, but the answers differ. The first answer is computed in line 9, and printed in line 10. The expression `seconds / factorInt` divides two integers, so the result is an integer and the remainder is lost. Even though the result is assigned to a `double` variable named `hours`, the division takes place first, so the remainder is lost before the assignment takes place. Since the value printed is a `double`, however, the output shows ".0" as the decimal portion of the value.

The second answer is computed in line 12 and printed in line 13. The expression `seconds / factorDouble` divides an integer by a `double`. The integer is promoted to a `double` before the expression is evaluated. The result is a `double` containing the usual 15, or so, digits of precision.

Note in line 7, a `double` variable `factorDouble` is declared and initialized with the constant `3600.0`. Strictly speaking, ".0" is not needed because the value `3600` (an `int` constant) would be promoted to a `double` anyway, as part of the assignment. It is good, however, to "show your intention" by appending ".0" to a whole number that is intended as a `double`. You can get into trouble by ignoring this practice when expressions involve two or more constants (as opposed to variables). For example, the expression `5 / 4` equals `1` (an `int`), whereas the expression `5 / 4.0` equals `1.25` (a `double`).

For simple assignment, an `int` can be assigned to a `double` (because of promotion); however, a `double` cannot be assigned to an `int`. In the following statements

```

int    x = 5
double y = 6.7;
y = x;           // OK! int is promoted to double
x = y;           // Wrong! can't assign a double to an int

```

the last line will cause a compiler error. If it is absolutely necessary to assign or convert an `int` to a `double`, "casting" is the answer. This is discussed in the next section.

## ***The cast Operator***

The *cast* operator explicitly converts one data type to another. Casting is useful, for example, where a conversion is necessary that does not automatically occur through promotion. Casting is also used to convert one object type to another, as we will meet later. The syntax for the cast operator is

```
(type)variable
```

or

```
(type)constant
```

The cast operator consists of the name of a data type enclosed in parentheses followed by a variable or constant. As a simple example, the following statements

```
double x = 3.14;  
int y = (int)x;  
System.out.println(y);
```

print 3 on the standard output. In the second line, the value of a `double` variable `x` is assigned to the `int` variable `y`. This is permitted because the `double` is cast to an `int` by preceding `x` with `"(int)"`. Note that a floating-point number is truncated (not rounded) when cast to an `int`.

At this point, you might be asking yourself, “Why is the conversion from one data type to another automatic in some cases, but only allowed through casting in other cases?” Anytime there is a potential “loss of information”, casting is required. The cast operator is like a safety check; it is your way of telling the compiler that you’re aware of the possible consequences of the conversion.

Let’s explore casting and promotion in more detail through an example program. Figure 6 contains the listing for `DemoPromoteAndCast.java`.

```

1 public class DemoPromoteAndCast
2 {
3     public static void main(String[] args)
4     {
5         // integer promotion
6         byte a = 1;
7         short b = a;           // byte promotes to short
8         int c = b;            // short promotes to int
9         long d = c;           // int promotes to long
10
11        // integer casting
12        c = (int)d;            // long casted to int
13        b = (short)c;          // int casted to short
14        a = (byte)b;           // short casted to byte
15
16        // floating-point promotion
17        float x = 1.0f;        // 'f' needed to indicate float
18        double y = x;          // float promotes to double
19
20        // character example
21        char aa = 'Q';
22        int xx = aa;           // character promotes to int
23        char bb = (char)xx;    // int casted to char
24
25        // floating-point casting
26        x = (float)y;          // double casted to float
27        c = (int)y;            // double casted to int
28    }
29 }

```

Figure 6. DemoPromoteAndCast.java

This program is as dull as they get. It receives no input and it generates no output. The key point is that it compiles without errors. All statements are perfectly legal Java statements. The objective is to exercise the rules of the Java language with respect to promotion and casting.

The program is divided into five sections. In lines 5-9, integer promotion is demonstrated, working from "lowest" to "highest". A byte variable, `a`, is declared in line 6 and assigned the value 1. In line 7, `a` is assigned to the short variable `b`. Since a byte is 8 bits and a short is 16 bits, a byte is considered a "lower" form of an integer. The assignment is legal because the lower form is automatically promoted to the higher form. A similar process is demonstrated in line 8 (promoting a short to an int) and line 9 (promoting an int to a long).

Integer casting is demonstrated in lines 11-14. Now we are working down the hierarchy of integer types, and this requires casting. Assigning a long to an int is permitted only if the long is cast to an int, as demonstrated in line 12. Similarly, an int can be assigned to a short by casting (line 13), and a short can be assigned to a byte by casting (line 14).

Java contains only two variations of floating-point numbers, float and double, and float (32 bits) is "lower" than double (64 bits). Assigning a float to a double is allowed (line 18), as automatic promotion occurs. In line 17, note that the constant assigned to the float variable `x` is coded as `1.0f`. A trailing 'f' or 'F' indicates a floating-point number is intended as a float. Appending 'd' or 'D' indicates double, but this is never needed since double is the default for floating-point numbers.

A character example appears in lines 20-23. A `char` variable `aa` is declared and assigned the constant `'Q'`. In line 23, the `char` variable `aa` is assigned to the `int` variable `xx`. This is permissible as the `char` is automatically promoted to an `int`. The value assigned is the Unicode value of the character.

Finally, floating-point casting is demonstrated in lines 25-27. First, a `double` is assigned to a `float` (line 26). Since `float` is the lower of the two types, casting is required. Finally, a `double` is assigned to an `int` through casting.

### **Increment, Decrement, Prefix, Postfix**

Adding or subtracting one (1) is so common in Java and other programming languages that a shorthand notation has evolved. For simple increment by 1 the `++` operator is used, and for simple decrement by 1, the `--` operator is used. So, the following two lines have the same effect:

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

as do

```
x = x - 1;
x--;
```

When these operators appear after the variable, as above, they are known as *postfix* operators. They can also appear in front of a variable as *prefix* operators. Although in both cases the effect is to increment or decrement the variable, there is a small but important difference. This difference is illustrated as follows:

```
int x = 5;
int y = 3 + x++;
```

After these statements execute, `y` equals 8 and `x` equals 6. Look carefully and see if you agree. Here's the operation: If a variable in an expression is bound with a postfix increment operator, the value used is the original value of the variable. The variable is incremented *after* the value is used in the expression. A similar statement can be made for the decrement operator. For these operators in the prefix position, the increment or decrement occurs *before* the variable is used in the expression. Prefix and postfix operators are summarized in Table 2.

Table 2. Prefix and Postfix Operators

| Expression       | Effect                        | Value of Expression |
|------------------|-------------------------------|---------------------|
| <code>x++</code> | increment <code>x</code> by 1 | <code>x</code>      |
| <code>++x</code> | increment <code>x</code> by 1 | <code>x + 1</code>  |
| <code>x--</code> | decrement <code>x</code> by 1 | <code>x</code>      |
| <code>--x</code> | decrement <code>x</code> by 1 | <code>x - 1</code>  |

### **Operation-Assignment Shorthand**

A popular shorthand notation has evolved for operators that normally take two arguments — one on the left, one on the right — such as the arithmetic operators. When these operators are combined with assignment, they can be coded using *op=* notation. For example, the following two statements have the same effect:

```
x = x + 3;  
x += 3;
```

as do

```
z = z / y;  
z /= y;
```

The *op=* notation is also valid for logical operators and bitwise operators, which we will meet soon.