Chapter 3

Arithmetic Expressions

Java programs can solve mathematical problems. Often programs that appear nonmathematical on the surface perform a large number of mathematical calculations behind the scenes. Graphical computer games are one example. The user experience with a game may be very different from that of solving algebra or trigonometry problems, but the executing program is continuously doing the mathematics behind the scenes in order to achieve the images displayed on the screen. This chapter examines some of Java's simpler mathematical capabilities.

3.1 Expressions and Mathematical Operators

A literal value (like 5) and a variable (like `x`) are simple *expressions*. Complex expressions can be built from simpler expressions using *operators*. Java supports the familiar arithmetic operators as show in Table 3.1. The arithmetic

Operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division
%	modulus (remainder)

Table 3.1: Arithmetic operators

operators show in Table 3.1 are called *binary operators* because they require two operands. The following interactive session illustrates addition:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> 2 + 2
4
> 10+4
14
> int x = 5;
> x + 3
8
```


Space around the + operator is optional but encouraged since it makes the expression easier to read.

The following session computes the circumference of a circle with a radius of 10 using the formula $C = 2\pi r$:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> final double PI = 3.14159;
> double radius = 10;
> double circumference = 2 * PI * radius;
> circumference
62.8318
```

π is a mathematical constant and should not vary; therefore, `PI` is declared `final` so its value cannot be changed accidentally later on during the session.

It is important to realize that the statement

```
double circumference = 2 * PI * radius;
```

is not definition of fact, but simply a computation and an assignment that is executed when encountered. To illustrate this concept, consider the following extended session:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> final double PI = 3.14159;
> double radius = 10;
> double circumference = 2 * PI * radius;
> circumference
62.8318
> radius = 4;
> circumference
62.8318
```

Here we see how even though `radius` changed, `circumference` was not automatically recomputed. This is because after `radius` was reassigned, `circumference` was *not* reassigned. Therefore, `circumference` must be recomputed if `radius` changes:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> final double PI = 3.14159;
> double radius = 10;
> double circumference = 2 * PI * radius;
> circumference
62.8318
> radius = 4;
> circumference = 2 * PI * radius;
> circumference
25.13272
```

As in algebra, multiplication and division have precedence over addition and subtraction, and parentheses can be used for grouping operations and overriding normal precedences.

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> 2 * 3 + 5
11
> 2 + 3 * 5
17
> 2 * (3 + 5)
16
> (2 + 3) * 5
25
```

In algebra, it is common to write the multiplication of two variables, $x \times y$, as xy. Mathematicians can do this because they use one letter variable names. Java allows and encourages longer variable names, so the multiplication operator (`*`) may not be omitted because if you have variables named `x` and `y`, `xy` is a new variable name. `x` times `y` must be written as `x * y`.

It may seem odd to include a remainder operator, but it actually quite useful. Consider the problem of computing the number of hours, minutes, and seconds in a given number of seconds. For example, how many hours, minutes, and seconds are there in 10,000 seconds? First let us develop some basic identities:

- 60 seconds = 1 minute
- 60 minutes = 1 hour

This means that one hour has 60×60 seconds, or 3,600 seconds. Armed with this information, we can do some calculations in the DrJava interpreter:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> int sec = 10000;
> int hr = sec/3600;
> sec = sec % 3600;
> int min = sec/60;
> sec = sec % 60;
> hr
2
> min
46
> sec
40
```

Thus 10,000 seconds equals 2 hours, 46 minutes, 40 seconds. Here is how the calculation works:

- `int sec = 10000;`

  Set the number of seconds to 10,000, the starting point of our computation.

- `int hr = sec/3600;`

  Since each hour has 3,600 seconds, we divide `sec` by 3,600. Note that Java you may not use commas to separate groups of digits. The result is assigned to the variable `hr`.

- `sec = sec % 3600;`

  Here we reassign `sec` to be the number of seconds remaining after the previous calculation that computed the hours. The previous calculation assigned `hr` but did not modify `sec`. This assignment statement actually modifies `sec`.

- `int min = sec/60;`
  `sec = sec % 60;`

  Similar to the previous item, we compute the number of minutes from the remaining seconds and then find the remaining seconds after the minutes computation, updating the seconds.

We can easily check our answer:

Interactions

```
> 2*60*60 + 46*60 + 40
10000
```

Java has two *unary* arithmetic operators. They are called unary operators because they have only one operand. The unary + has no effect on its operand, while unary − returns the additive inverse of its operand:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> 2 + 3 * 4
14
> +(2 + 3 * 4)
14
> -(2 + 3 * 4)
-14
1
> x = 5;
> -x
-5
```

3.2 String Concatenation

Strings can be spliced together (a process known as *concatenation*) using the + operator. Consider:

Interactions

```
> "Part 1" + "Part2"
"Part 1Part2"
```

The compiler and interpreter will automatically convert a primitive type into a human-readable string when required; for example, when a string is concatenated with a primitive type:

Interactions

```
> "Part " + 1
"Part 1"
> "That is " + true
"That is true"
```


Notice that neither 1 nor `true` are strings but are, respectively, `int` and `boolean` literals. We say the `+` operator is *overloaded* since it performs two distinct functions:

- When both operands are numbers, `+` performs familiar arithmetic addition.
- When one or both of its operands is a `String`, `+` converts the non-`String` operand to a `String` if necessary and then performs string concatenation.

String variables can be concatenated just like string literals:

Interactions

```
Welcome to DrJava.  Working directory is /Users/rick/java
> String phrase = "The time is now";
> int hour = 4, minute = 12;
> String time = phrase + " " + hour + ":" + minute + " exactly.";
> time
"The time is now 4:12 exactly."
```

3.3 Arithmetic Abbreviations

Expressions do not affect the value of variables they contain. The expression

```
x + 1
```

evaluates to one more than `x`'s value, but `x`'s value is not changed. It is misleading to say the expression "adds one to `x`," since this implies `x` becomes one more than before the expression was evaluated. An assignment operator actually changes a variable's value. We have seen the simple assignment operator, `=`. Several more assignment operators exist that allow arithmetic to be combined with assignment. The statements

```
x = x + 1;
y = y * 2;
z = z - 3;
w = w / 4;
t = t % 5;
```

can all be expressed in a different, shorter way:

```
x += 1;
y *= 2;
z -= 3;
w /= 4;
t %= 5;
```

In all the arithmetic-assignment operators, no space can come between the arithmetic symbol and the equals symbol.

The task of increasing or decreasing a variable's value is common in computing. Java provides the following shortcut:


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

can be rewritten as:

```
x++;
```

or as:

```
++x;
```

There is a slight different between the two notations as an exercise illustrates. When used as a standalone statements as shown here, the preincrement (`++x`) and postincrement (`x++`) versions work the same. Similarly,

```
x--;
```

and

```
--x;
```

decrease `x`'s value by one. Note how the `++` and `--` operators *are* assignment operators even though no `=` symbol appears in the operator. As with the arithmetic-assignment operators, no space may appear between the symbols that make up the `++` and `--` operators. The `++` and `--` variables can only be applied to numeric variables, not expressions and not nonnumeric variables.

3.4 Summary

- The arithmetic operators `+`, `-`, `*`, `/`, and `%` perform mathematical operations on variables and values.
- In the expression *x* + *y*, `+` is the operator, while *x* and *y* are its operands.
- The unary `+` and `-` require only a single operand.
- Arithmetic expressions mixing `+`, `-`, `*`, and/or `/` are evaluated in the same order as they are in normal arithmetic (multiplication before addition, etc.).
- As in standard arithmetic, parentheses can be used to override the normal order of evaluation.
- The `+` operator can be used to concatenate two strings. Any nonstring operands are converted into strings.
- Java's statements involving assignment with simple arithmetic modification have convenient abbreviations.

3.5 Exercises

1. How does the `/` operator differ from the operator? Provide an example illustrating the difference.
2. Evaluate each of the following Java expressions:

a. `2 + 2`	b. `2 % 2`	c. `5 / 3`	d. `5 % 3`	e. `5.0 / 3.0`
f. `5.0 / 3`	g. `5 / 3.0`	h. `(1 + 2) * 5`	i. `(1 * 2) + 5`	j. `1 + 2 * 5`
k. `1 * 2 + 5`	l. `--4`	m.	n.	o.

3. Is it legal for the same variable name to appear on both sides of an assignment statement? How is this useful? What is the restriction on the left-hand side of the statement?

4. How is an expression different from a statement?

5. What is concatenation? On what type is concatenation performed?

6. What happens when a string is concatenated with an `int` value? What happens when a string is concatenated with an `int` variable?

7. How are unary operators different from binary operators?

8. If necessary consult a mathematics book to review the basic properties of arithmetic operations, then answer the following questions about Java's arithmetic operations:

   a. Is + commutative over `int`s? b. Is + associative over `int`s? c. Is * commutative over `int`s?
   f. Does the distributive property hold for * and +? g. Is / closed over `int`s? h.
   k. l. m.

9. What is the difference in behavior of the following two statements:

```
x += 2;
x =+ 2;
```

   and the following two statements:

```
x -= 5;
x =- 5;
```

   and the following two statements:

```
x *= 3;
x =* 3;
```

   Explain the results.

10. Set up the situation where you can use the following statements in the interpreter:

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

   and

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

   Explain why it really does make a difference whether the ++ comes before or after the variable being incremented.

   Does your theory hold for the -- operator?

