5.4 Examples Using the `for` Statement

The following examples show techniques for varying the control variable in a `for` statement. In each case, we write *only* the appropriate `for` header. Note the change in the relational operator for the loops that *decrement* the control variable.

1. Vary the control variable from `1` to `100` in increments of `1`.

```
for (int i = 1; i <= 100; i++)
```

2. Vary the control variable from `100` to `1` in *decrements* of `1`.

```
for (int i = 100; i >= 1; i--)
```

3. Vary the control variable from `7` to `77` in increments of `7`.

```
for (int i = 7; i <= 77; i += 7)
```

4. Vary the control variable from `20` to `2` in *decrements* of `2`.

```
for (int i = 20; i >= 2; i -= 2)
```

5. Vary the control variable over the values `2`, `5`, `8`, `11`, `14`, `17`, `20`.

```
for (int i = 2; i <= 20; i += 3)
```

6. Vary the control variable over the values `99`, `88`, `77`, `66`, `55`, `44`, `33`, `22`, `11`, `0`.

```
for (int i = 99; i >= 0; i -= 11)
```

Common Programming Error 5.5

Using an incorrect relational operator in the loop-continuation condition of a loop that counts downward (e.g., using `i <= 1` *instead of* `i >= 1` *in a loop counting down to 1) is usually a logic error.*

Common Programming Error 5.6

Do not use equality operators (`!=` *or* `==`*) in a loop-continuation condition if the loop's control variable increments or decrements by more than 1. For example, consider the* `for` *statement header* `for (int counter = 1; counter != 10; counter += 2)`. *The loop-continuation test* `counter != 10` *never becomes* `false` *(resulting in an* infinite loop*) because* `counter` *increments*

by 2 after each iteration.

Error-Prevention Tip 5.6

Counting loops are error prone. In subsequent chapters, we'll introduce lambdas and streams—technologies that you can use to eliminate such errors.

5.4.1 Application: Summing the Even Integers from 2 to 20

We now consider two sample applications that demonstrate simple uses of `for`. The application in Fig. 5.5 uses a `for` statement to sum the even integers from 2 to 20 and store the result in an `int` variable called `total`.

```
1   // Fig. 5.5: Sum.java
2   // Summing integers with the for statement.
3
4   public class Sum {
5      public static void main(String[] args) {
6         int total = 0;
7
8         // total even integers from 2 through 20
9         for (int number = 2; number <= 20; number
10           total += number;
11
```

```
                    12         }
   13         System.out.printf("Sum is %d%n", total);
                    14      }
                     15  }
```

```
Sum is 110
```

Fig. 5.5

Summing integers with the for statement.

The *initialization* and *increment* expressions can be comma-separated lists that enable you to use multiple initialization expressions or multiple increment expressions. For example, *although this is discouraged*, you could merge the `for` statement's body in lines 9–11 of Fig. 5.5 into the increment portion of the `for` header by using a comma as follows:

```
for (int number = 2; number <= 20; total += number, n
    ; // empty statement
}
```

Good Programming Practice 5.1

For readability limit the size of control-statement headers to a single line if possible.

5.4.2 Application: Compound-Interest Calculations

Next, let's use the `for` statement to compute compound interest. Consider the following problem:

A person invests $1,000 in a savings account yielding 5% interest. Assuming that all the interest is left on deposit, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula to determine the amounts:

$$a = p(1 + r)^n$$

where

p is the original amount invested (i.e., the principal)

r is the annual interest rate (e.g., use 0.05 for 5%)

n is the number of years

a is the amount on deposit at the end of the nth year.

The solution to this problem (Fig. 5.6) involves a loop that performs the indicated calculation for each of the 10 years the

money remains on deposit. Lines 6, 7 and 15 in `main` declare `double` variables `principal`, `rate` and `amount`. Lines 6–7 initialize `principal`to `1000.0` and `rate` to `0.05`. Line 15 initializes `amount` to the result of the compound-interest calculation. Java treats floating-point constants like `1000.0` and `0.05` as type `double`. Similarly, Java treats whole-number constants like `7` and `-22` as type `int`.

```
1   // Fig. 5.6: Interest.java
2   // Compound-interest calculations with for.
3
4   public class Interest {
5      public static void main(String[] args) {
6         double principal = 1000.0; // initial amou
7         double rate = 0.05; // interest rate
8
9         // display headers
10        System.out.printf("%s%20s%n", "Year", "Amou
11
12     // calculate amount on deposit for each of te
13        for (int year = 1; year <= 10; ++year) {
14           // calculate new amount on deposit for spe
15           double amount = principal * Math.pow(1.0 +
16
17           // display the year and the amount
18           System.out.printf("%4d%,20.2f%n", year, am
19        }
20     }
21  }
```

Year	Amount on deposit
1	1,050.00
2	1,102.50

3	1,157.63
4	1,215.51
5	1,276.28
6	1,340.10
7	1,407.10
8	1,477.46
9	1,551.33
10	1,628.89

Fig. 5.6

Compound-interest calculations with for.

Formatting Strings with Field Widths and Justification

Line 10 outputs the headers for two columns of output. The first column displays the year and the second column the amount on deposit at the end of that year. We use the format specifier `%20s` to output the `String "Amount on Deposit"`. The integer `20` between the `%` and the conversion character `s` indicates that the value should be displayed with a **field width** of 20—that is, `printf` displays the value with at

least 20 character positions. If the value to be output is less than 20 character positions wide (17 characters in this example), the value is **right justified** in the field by default. If the `year` value to be output were more than four character positions wide, the field width would be extended to the right to accommodate the entire value—this would push the `amount` field to the right, upsetting the neat columns of our tabular output. To output values **left justified**, simply precede the field width with the **minus sign (–) formatting flag** (e.g., `%-20s`).

Performing the Interest Calculations with `static` Method `pow` of Class `Math`

The `for` statement (lines 13–19) executes its body 10 times, varying control variable `year` from 1 to 10 in increments of 1. This loop terminates when `year` becomes 11. (Variable `year` represents *n* in the problem statement.)

Classes provide methods that perform common tasks on objects. In fact, most methods must be called on a specific object. For example, to output text in Fig. 5.6, line 10 calls method `printf` on the `System.out` object. Some classes also provide methods that perform common tasks and do *not* require you to first create objects of those classes. These are called `static` methods. For example, Java does not include an exponentiation operator, so the designers of Java's `Math`

class defined `static` method `pow` for raising a value to a power. You can call a `static` method by specifying the *class name* followed by a dot (`.`) and the method name, as in

```
ClassName.methodName(arguments)
```

In Chapter 6, you'll learn how to implement `static` methods in your own classes.

We use `static` method `pow` of class `Math` to perform the compound-interest calculation in Fig. 5.6. `Math.pow(`*x*`, `*y*`)` calculates the value of *x* raised to the *y*th power. The method receives two `double` arguments and returns a `double` value. Line 15 performs the calculation $a = p(1 + r)^n$, where *a* is `amount`, *p* is `principal`, *r* is `rate` and *n* is `year`. Class `Math` is defined in package `java.lang`, so you do *not* need to import class `Math` to use it.

The body of the `for` statement contains the calculation `1.0 + rate`, which appears as an argument to the `Math.pow` method. In fact, this calculation produces the *same* result each time through the loop, so repeating it in every iteration of the loop is wasteful.

Performance Tip 5.1

In loops, avoid calculations for which the result never changes—such calculations should typically be placed before the loop.

Many of today's sophisticated optimizing compilers will place such calculations outside loops in the compiled code.

Formatting Floating-Point Numbers

After each calculation, line 18 outputs the year and the amount on deposit at the end of that year. The year is output in a field width of four characters (as specified by `%4d`). The amount is output as a floating-point number with the format specifier `%,20.2f`.

The **comma (`,`) formatting flag** indicates that the floating-point value should be output with a **grouping separator**. The actual separator used is specific to the user's locale (i.e., country). For example, in the United States, the number will be output using commas to separate every three digits and a decimal point to separate the fractional part of the number, as in 1,234.45. The number `20` in the format specification indicates that the value should be output right justified in a *field width* of 20 characters. The `.2` specifies the formatted number's *precision*—in this case, the number is *rounded* to the nearest hundredth and output with two digits to the right of the decimal point.

A Warning about Displaying

Rounded Values

We declared variables `amount`, `principal` and `rate` to be of type `double` in this example. We're dealing with fractional parts of dollars and thus need a type that allows decimal points in its values. Unfortunately, floating-point numbers can cause trouble. Here's a simple explanation of what can go wrong when using `double` (or `float`) to represent dollar amounts (assuming that dollar amounts are displayed with two digits to the right of the decimal point): Two `double` dollar amounts stored in the machine could be 14.234 (which would normally be rounded to 14.23 for display purposes) and 18.673 (which would normally be rounded to 18.67 for display purposes). When these amounts are added, they produce the internal sum 32.907, which would normally be rounded to 32.91 for display purposes. Thus, your output could appear as

```
  14.23
+ 18.67
  32.91
```

but a person adding the individual numbers as displayed would expect the sum to be 32.90. You've been warned!

Error-Prevention Tip 5.7

Do not use variables of type `double` *(or* `float`*) to perform precise monetary calculations. The imprecision of floating-*

point numbers can lead to errors. In the exercises, you'll learn how to use integers to perform precise monetary calculations —Java also provides class `java.math.BigDecimal` *for this purpose, which we demonstrate in Fig. 8.16.*

Error-Prevention Tip 5.8

In a global economy, dealing with currencies, monetary amounts, conversions, rounding and formatting is complex. The new JavaMoney API (`http://javamoney.github.io`*) was developed to meet these challenges. At the time of this writing, it was not yet incorporated into the JDK. Chapter 8 suggests a JavaMoney project exercise.*