Chapter 2

Primitive Data and Definite Loops

Introduction

Now that you know something about the basic structure of Java programs, you are ready to learn how to solve more complex problems. For the time being we will still concentrate on programs that produce output, but we will begin to explore some of the aspects of programming that require problem-solving skills.

The first half of this chapter fills in two important areas. First, it examines expressions, which are used to perform simple computations in Java, particularly those involving numeric data. Second, it discusses program elements called variables that can change in value as the program executes.

The second half of the chapter introduces your first control structure: the `for` loop. You use this structure to repeat actions in a program. This is useful whenever you find a pattern in a task such as the creation of a complex figure, because you can use a `for` loop to repeat the action to create that particular pattern. The challenge is finding each pattern and figuring out what repeated actions will reproduce it.

The `for` loop is a flexible control structure that can be used for many tasks. In this chapter we use it for *definite loops*, where you know exactly how many times you want to perform a particular task. In Chapter 5 we will discuss how to write *indefinite loops*, where you don't know in advance how many times to perform a task.

2.1 Basic Data Concepts

Programs manipulate information, and information comes in many forms. Java is a *type-safe* language, which means that it requires you to be explicit about what kind of information you intend to manipulate and it guarantees that you manipulate the data in a reasonable manner. Everything that you manipulate in a Java program will be of a certain *type,* and you will constantly find yourself telling Java what types of data you intend to use.

> **Data Type**
>
> A name for a category of data values that are all related, as in type `int` in Java, which is used to represent integer values.

A decision was made early in the design of Java to support two different kinds of data: primitive data and objects. The designers made this decision purely on the basis of performance, to make Java programs run faster. Unfortunately, it means that you have to learn two sets of rules about how data works, but this is one of those times when you simply have to pay the price if you want to use an industrial-strength programming language. To make things a little easier, we will study the primitive data types first, in this chapter; in the next chapter, we will turn our attention to objects.

Primitive Types

There are eight primitive data types in Java: `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`. Four of these are considered fundamental: `boolean`, `char`, `double`, and `int`. The other four types are variations that exist for programs that have special requirements. The four fundamental types that we will explore are listed in Table 2.1.

The type names (`int`, `double`, `char`, and `boolean`) are Java keywords that you will use in your programs to let the compiler know that you intend to use that type of data.

It may seem odd to use one type for integers and another type for real numbers. Isn't every integer a real number? The answer is yes, but these are fundamentally different types of numbers. The difference is so great that we make this distinction even in English. We don't ask, "How much sisters do you have?" or "How many do you weigh?" We realize that sisters come in discrete integer quantities (0 sisters, 1 sister, 2 sisters, 3 sisters, and so on), and we use the word "many" for integer quantities ("How

Table 2.1 Commonly Used Primitive Types in Java

Type	Description	Examples
`int`	integers (whole numbers)	`42, -3, 18, 20493, 0`
`double`	real numbers	`7.35, 14.9, -19.83423`
`char`	single characters	`'a', 'X', '!'`
`boolean`	logical values	`true, false`

many sisters do you have?"). Similarly, we realize that weight can vary by tiny amounts (175 pounds versus 175.5 pounds versus 175.25 pounds, and so on), and we use the word "much" for these real-number quantities ("How much do you weigh?").

In programming, this distinction is even more important, because integers and reals are represented in different ways in the computer's memory: Integers are stored exactly, while reals are stored as approximations with a limited number of digits of accuracy. You will see that storing values as approximations can lead to round-off errors when you use real values.

The name `double` for real values is not very intuitive. It's an accident of history in much the same way that we still talk about "dialing" a number on our telephones even though modern telephones don't have dials. The C programming language introduced a type called `float` (short for "floating-point number") for storing real numbers. But `float`s had limited accuracy, so another type was introduced, called `double` (short for "double precision," meaning that it had double the precision of a simple `float`). As memory became cheaper, people began using `double` as the default for floating-point values. In hindsight, it might have been better to use the word `float` for what is now called `double` and a word like "half" for the values with less accuracy, but it's tough to change habits that are so ingrained. So, programming languages will continue to use the word `double` for floating-point numbers, and people will still talk about "dialing" people on the phone even if they've never touched a telephone dial.

Expressions


VideoNote

When you write programs, you will often need to include values and calculations. The technical term for these elements is *expressions*.

Expression

A simple value or a set of operations that produces a value.

The simplest expression is a specific value, like `42` or `28.9`. We call these "literal values," or *literals*. More complex expressions involve combining simple values. Suppose, for example, that you want to know how many bottles of water you have. If you have two 6-packs, four 4-packs, and two individual bottles, you can compute the total number of bottles with the following expression:

```
(2 * 6) + (4 * 4) + 2
```

Notice that we use an asterisk to represent multiplication and that we use parentheses to group parts of the expression. The computer determines the value of an expression by *evaluating* it.

Evaluation

The process of obtaining the value of an expression.

The value obtained when an expression is evaluated is called the *result.* Complex expressions are formed using *operators.*

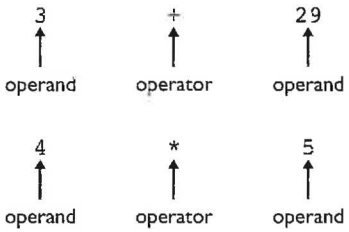
Operator

A special symbol (like + or *) that is used to indicate an operation to be performed on one or more values.

The values used in the expression are called *operands.* For example, consider the following simple expressions:

```
3 + 29
4 * 5
```

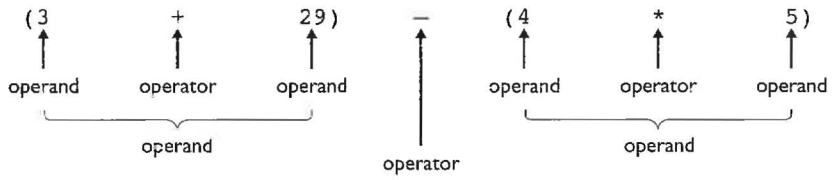
The operators here are the + and *, and the operands are simple numbers.



When you form complex expressions, these simpler expressions can in turn become operands for other operators. For example, the expression

```
(3 + 29) - (4 * 5)
```

has two levels of operators.



The addition operator has simple operands of `3` and `29` and the multiplication operator has simple operands of `4` and `5`, but the subtraction operator has operands that are each parenthesized expressions with operators of their own. Thus, complex expressions can be built from smaller expressions. At the lowest level, you have simple numbers. These are used as operands to make more complex expressions, which in turn can be used as operands in even more complex expressions.

There are many things you can do with expressions. One of the simplest things you can do is to print the value of an expression using a `println` statement. For example, if you say:

```
System.out.println(42);
System.out.println(2 + 2);
```

you will get the following two lines of output:

```
42
4
```

Notice that for the second `println`, the computer evaluates the expression (adding 2 and 2) and prints the result (in this case, 4).

You will see many different operators as you progress through this book, all of which can be used to form expressions. Expressions can be arbitrarily complex, with as many operators as you like. For that reason, when we tell you, "An expression can be used here," we mean that you can use arbitrary expressions that include complex expressions as well as simple values.

Literals

The simplest expressions refer to values directly using what are known as *literals*. An integer literal (considered to be of type `int`) is a sequence of digits with or without a leading sign:

```
3    482    -29434    0    92348    +9812
```

A floating-point literal (considered to be of type `double`) includes a decimal point:

```
298.4    0.284    207.    .2843    -17.452    -.98
```

Notice that `207.` is considered a `double` even though it coincides with an integer, because of the decimal point. Literals of type `double` can also be expressed in scientific notation (a number followed by `e` followed by an integer):

```
2.3e4    1e-5    3.84e92    2.458e12
```

The first of these numbers represents 2.3 times 10 to the 4th power, which equals 23,000. Even though this value happens to coincide with an integer, it is considered to be of type `double` because it is expressed in scientific notation. The second number represents 1 times 10 to the –5th power, which is equal to 0.00001. The third number represents 3.84 times 10 to the 92nd power. The fourth number represents 2.458 times 10 to the 12th power.

We have seen that textual information can be stored in literal strings that store a sequence of characters. In later chapters we will explore how to process a string

character by character. Each such character is of type `char`. A character literal is enclosed in single quotation marks and includes just one character:

```
'a'    'm'    'X'    '!'    '3'    '\\'
```

All of these examples are of type `char`. Notice that the last example uses an escape sequence to represent the backslash character. You can even refer to the single quotation character using an escape sequence:

```
'\''
```

Finally, the primitive type `boolean` stores logical information. We won't be exploring the use of type `boolean` until we reach Chapter 4 and see how to introduce logical tests into our programs, but for completeness, we include the `boolean` literal values here. Logic deals with just two possibilities: true and false. These two Java keywords are the two literal values of type `boolean`:

```
true    false
```

Arithmetic Operators

The basic arithmetic operators are shown in Table 2.2. The addition and subtraction operators will, of course, look familiar to you, as should the asterisk as a multiplication operator and the forward slash as a division operator. However, as you'll see, Java has two different division operations. The remainder or mod operation may be unfamiliar.

Division presents a problem when the operands are integers. When you divide 119 by 5, for example, you do not get an integer result. Therefore, the results of integer division are expressed as two different integers, a quotient and a remainder:

$$\frac{119}{5} = 23 \text{ (quotient) with } 4 \text{ (remainder)}$$

In terms of the arithmetic operators:

`119 / 5` evaluates to `23`
`119 % 5` evaluates to `4`

Table 2.2 Arithmetic Operators in Java

Operator	Meaning	Example	Result
+	addition	`2 + 2`	4
–	subtraction	`53 - 18`	35
*	multiplication	`3 * 8`	24
/	division	`4.8 / 2.0`	2.4
%	remainder or mod	`19 % 5`	4

These two division operators should be familiar if you recall how long-division calculations are performed:

```
      31
34)1079
   102
   ---
    59
    34
    --
    25
```

Here, dividing 1079 by 34 yields 31 with a remainder of 25. Using arithmetic operators, the problem would be described like this:

`1079 / 34` evaluates to `31`
`1079 % 34` evaluates to `25`

It takes a while to get used to integer division in Java. When you are using the division operator (`/`), the key thing to keep in mind is that it truncates anything after the decimal point. So, if you imagine computing an answer on a calculator, just think of ignoring anything after the decimal point:

- `19/5` is `3.8` on a calculator, so `19/5` evaluates to `3`
- `207/10` is `20.7` on a calculator, so `207/10` evaluates to `20`
- `3/8` is `0.375` on a calculator, so `3/8` evaluates to `0`

The remainder operator (`%`) is usually referred to as the "mod operator," or simply "mod." The mod operator lets you know how much was left unaccounted for by the truncating division operator. For example, given the previous examples, you'd compute the mod results as shown in Table 2.3.

In each case, you figure out how much of the number is accounted for by the truncating division operator. The mod operator gives you any excess (the remainder). When you put this into a formula, you can think of the mod operator as behaving as follows:

```
x % y = x - (x / y) * y
```

Table 2.3 Examples of Mod Operator

Mod problem	First divide	What does division account for?	How much is left over?	Answer
`19 % 5`	19/5 is 3	3 * 5 is 15	19 − 15 is 4	4
`207 % 10`	207/10 is 20	20 * 10 is 200	207 − 200 is 7	7
`3 % 8`	3/8 is 0	0 * 8 is 0	3 − 0 is 3	3

It is possible to get a result of `0` for the mod operator. This happens when one number divides evenly into another. For example, each of the following expressions evaluates to `0` because the second number goes evenly into the first number:

```
28 % 7
95 % 5
44 % 2
```

A few special cases are worth noting because they are not always immediately obvious to novice programmers:

- **Numerator smaller than denominator:** In this case division produces 0 and mod produces the original number. For example, `7 / 10` is `0` and `7 % 10` is `7`.
- **Numerator of 0:** In this case both division and mod return 0. For example, both `0 / 10` and `0 % 10` evaluate to `0`.
- **Denominator of 0:** In this case, both division and mod are undefined and produce a runtime error. For example, a program that attempts to evaluate either `7 / 0` or `7 % 0` will throw an `ArithmeticException` error.

The mod operator has many useful applications in computer programs. Here are just a few ideas:

- Testing whether a number is even or odd (`number % 2` is `0` for evens, `number % 2` is `1` for odds).
- Finding individual digits of a number (e.g., `number % 10` is the final digit).
- Finding the last four digits of a social security number (`number % 10000`).

The remainder operator can be used with `doubles` as well as with integers, and it works similarly: You consider how much is left over when you take away as many "whole" values as you can. For example, the expression `10.2 % 2.4` evaluates to `0.6` because you can take away four `2.4`s from `10.2`, leaving you with `0.6` left over.

For floating-point values (values of type `double`), the division operator does what we consider "normal" division. So, even though the expression `119 / 5` evaluates to `23`, the expression `119.0 / 5.0` evaluates to `23.8`.

Precedence

Java expressions are like complex noun phrases in English. Such phrases are subject to ambiguity. For example, consider the phrase "the man on the hill by the river with the telescope." Is the river by the hill or by the man? Is the man holding the telescope, or is the telescope on the hill, or is the telescope in the river? We don't know how to group the various parts together.

You can get the same kind of ambiguity if parentheses aren't used to group the parts of a Java expression. For example, the expression `2 + 3 * 4` has two operators. Which operation is performed first? You could interpret this two ways:

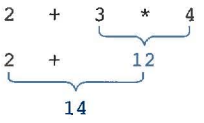
The first of these evaluates to `20` while the second evaluates to `14`. To deal with the ambiguity, Java has rules of *precedence* that determine how to group together the various parts.

Precedence

The binding power of an operator, which determines how to group parts of an expression.

The computer applies rules of precedence when the grouping of operators in an expression is ambiguous. An operator with high precedence is evaluated first, followed by operators of lower precedence. Within a given level of precedence the operators are evaluated in one direction, usually left to right.

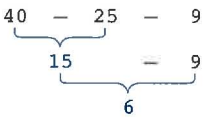
For arithmetic expressions, there are two levels of precedence. The multiplicative operators (`*`, `/`, `%`) have a higher level of precedence than the additive operators (+, –). Thus, the expression `2 + 3 * 4` is interpreted as



Within the same level of precedence, arithmetic operators are evaluated from left to right. This often doesn't make a difference in the final result, but occasionally it does. Consider, for example, the expression

```
40 - 25 - 9
```

which evaluates as follows:



You would get a different result if the second subtraction were evaluated first.

You can always override precedence with parentheses. For example, if you really want the second subtraction to be evaluated first, you can force that to happen by introducing parentheses:

```
40 - (25 - 9)
```

Table 2.4 Java Operator Precedence

Description	Operators
unary operators	+, –
multiplicative operators	*, /, %
additive operators	+, –

The expression now evaluates as follows:

```
40 – (25 – 9)
40 –     16
     24
```

Another concept in arithmetic is *unary* plus and minus, which take a single operand, as opposed to the binary operators we have seen thus far (e.g., *, /, and even binary + and –), all of which take two operands. For example, we can find the negation of `8` by asking for `–8`. These unary operators have a higher level of precedence than the multiplicative operators. Consequently, we can form expressions like the following:

```
12 * –8
```

which evaluates to `–96`.

We will see many types of operators in the next few chapters. Table 2.4 is a precedence table that includes the arithmetic operators. As we introduce more operators, we'll update this table to include them as well. The table is ordered from highest precedence to lowest precedence and indicates that Java will first group parts of an expression using the unary operators, then using the multiplicative operators, and finally using the additive operators.

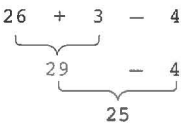
Before we leave this topic, let's look at a complex expression and see how it is evaluated step by step. Consider the following expression:

```
13 * 2 + 239 / 10 % 5 – 2 * 2
```

It has a total of six operators: two multiplications, one division, one mod, one subtraction, and one addition. The multiplication, division, and mod operations will be performed first, because they have higher precedence, and they will be performed from left to right because they are all at the same level of precedence:

```
13 * 2 + 239 / 10 % 5 – 2 * 2
  26   + 239 / 10 % 5 – 2 * 2
  26   +     23   % 5 – 2 * 2
  26   +          3   – 2 * 2
  26   +          3   –     4
```

Now we evaluate the additive operators from left to right:



Mixing Types and Casting

You'll often find yourself mixing values of different types and wanting to convert from one type to another. Java has simple rules to avoid confusion and provides a mechanism for requesting that a value be converted from one type to another.

Two types that are frequently mixed are `int`s and `double`s. You might, for example, ask Java to compute `2 * 3.6`. This expression includes the `int` literal `2` and the `double` literal `3.6`. In this case, Java converts the `int` into a `double` and performs the computation entirely with `double` values; this is always the rule when Java encounters an `int` where it was expecting a `double`.

This becomes particularly important when you form expressions that involve division. If the two operands are both of type `int`, Java will use integer (truncating) division. If either of the two operands is of type `double`, however, it will do real-valued (normal) division. For example, `23 / 4` evaluates to `5`, but all of the following evaluate to `5.75`:

```
23.0 / 4
23. / 4
23 / 4.0
23 / 4.
23. / 4.
23.0 / 4.0
```

Sometimes you want Java to go the other way, converting a `double` into an `int`. You can ask Java for this conversion with a *cast*. Think of it as "casting a value in a different light." You request a cast by putting the name of the type you want to cast to in parentheses in front of the value you want to cast. For example,

```
(int) 4.75
```

will produce the `int` value `4`. When you cast a `double` value to an `int`, it simply truncates anything after the decimal point.

If you want to cast the result of an expression, you have to be careful to use parentheses. For example, suppose that you have some books that are each 0.15 feet wide and you want to know how many of them will fit in a bookshelf that is 2.5 feet wide. You could do a straight division of `2.5 / 0.15`, but that evaluates to a `double` result that is between 16 and 17. Americans use the phrase "16 and change" as a way to express the idea that a value is larger than 16 but not as big as 17. In this case, we

don't care about the "change"; we only want to compute the 16 part. You might form the following expression:

```
(int) 2.5 / 0.15
```

Unfortunately, this expression evaluates to the wrong answer because the cast is applied to whatever comes right after it (here, the value `2.5`). This casts `2.5` into the integer `2`, divides by `0.15`, and evaluates to `13` and change, which isn't an integer and isn't the right answer. Instead, you want to form this expression:

```
(int) (2.5 / 0.15)
```

This expression first performs the division to get 16 and change, and then casts that value to an `int` by truncating it. It thus evaluates to the `int` value `16`, which is the answer you're looking for.

2.2 Variables


VideoNote

Primitive data can be stored in the computer's memory in a *variable.*

Variable

A memory location with a name and a type that stores a value.

Think of the computer's memory as being like a giant spreadsheet that has many cells where data can be stored. When you create a variable in Java, you are asking it to set aside one of those cells for this new variable. Initially the cell will be empty, but you will have the option to store a value in the cell. And as with a spreadsheet, you will have the option to change the value in that cell later.

Java is a little more picky than a spreadsheet, though, in that it requires you to tell it exactly what kind of data you are going to store in the cell. For example, if you want to store an integer, you need to tell Java that you intend to use type `int`. If you want to store a real value, you need to tell Java that you intend to use a `double`. You also have to decide on a name to use when you want to refer to this memory location. The normal rules of Java identifiers apply (the name must start with a letter, which can be followed by any combination of letters and digits). The standard convention in Java is to start variable names with a lowercase letter, as in `number` or `digits`, and to capitalize any subsequent words, as in `numberOfDigits`.

To explore the basic use of variables, let's examine a program that computes an individual's *body mass index* (BMI). Health professionals use this number to advise people about whether or not they are overweight. Given an individual's height and weight, we can compute that person's BMI. A simple BMI program, then, would naturally have three variables for these three pieces of information. There are several details that we need to discuss about variables, but it can be helpful to look at a complete

program first to see the overall picture. The following program computes and prints the BMI for an individual who is 5 feet 10 inches tall and weighs 195 pounds:

```
1  public class BMICalculator {
2      public static void main(String[] args) {
3          // declare variables
4          double height;
5          double weight;
6          double bmi;
7
8          // compute BMI
9          height = 70;
10         weight = 195;
11         bmi = weight / (height * height) * 703;
12
13         // print results
14         System.out.println("Current BMI:");
15         System.out.println(bmi);
16     }
17 }
```

Notice that the program includes blank lines to separate the sections and comments to indicate what the different parts of the program do. It produces the following output:

```
Current BMI:
27.976530612244897
```

Let's now examine the details of this program to understand how variables work. Before variables can be used in a Java program, they must be declared. The line of code that declares the variable is known as a variable *declaration.*

Declaration

A request to set aside a new variable with a given name and type.

Each variable is declared just once. If you declare a variable more than once, you will get an error message from the Java compiler. Simple variable declarations are of the form

```
<type> <name>;
```

as in the three declarations at the beginning of our sample program:

```
double height;
double weight;
double bmi;
```

Notice that a variable declaration, like a statement, ends with a semicolon. These declarations can appear anywhere a statement can occur. The declaration indicates the type and the name of the variable. Remember that the name of each primitive type is a keyword in Java (`int`, `double`, `char`, `boolean`). We've used the keyword `double` to define the type of these three variables.

Once a variable is declared, Java sets aside a memory location to store its value. However, with the simple form of variable declaration used in our program, Java does not store initial values in these memory locations. We refer to these as *uninitialized* variables, and they are similar to blank cells in a spreadsheet:

So how do we get values into those cells? The easiest way to do so is using an *assignment statement.* The general syntax of the assignment statement is

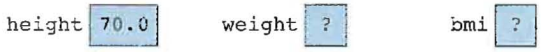
```
<variable> = <expression>;
```

as in

```
height = 70;
```

This statement stores the value `70` in the memory location for the variable `height`, indicating that this person is 70 inches tall (5 feet 10 inches). We often use the phrase "gets" or "is assigned" when reading a statement like this, as in "`height` gets `70`" or "`height` is assigned `70`."

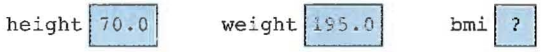
When the statement executes, the computer first evaluates the expression on the right side; then, it stores the result in the memory location for the given variable. In this case the expression is just a simple literal value, so after the computer executes this statement, the memory looks like this:



Notice that the value is stored as `70.0` because the variable is of type `double`. The variable `height` has now been initialized, but the variables `weight` and `bmi` are still uninitialized. The second assignment statement gives a value to `weight`:

```
weight = 195;
```

After executing this statement, the memory looks like this:



The third assignment statement includes a formula (an expression to be evaluated):

```
bmi = weight / (height * height) * 703;
```

To calculate the value of this expression, the computer divides the weight by the square of the height and then multiplies the result of that operation by the literal value `703`. The result is stored in the variable `bmi`. So, after the computer has executed the third assignment statement, the memory looks like this:

The last two lines of the program report the BMI result using `println` statements:

```
System.out.println("Current BMI:");
System.out.println(bmi);
```

Notice that we can include a variable in a `println` statement the same way that we include literal values and other expressions to be printed.

As its name implies, a variable can take on different values at different times. For example, consider the following variation of the BMI program, which computes a new BMI assuming the person lost 15 pounds (going from 195 pounds to 180 pounds).

```
 1  public class BMICalculator2 {
 2      public static void main(String[] args) {
 3          // declare variables
 4          double height;
 5          double weight;
 6          double bmi;
 7
 8          // compute BMI
 9          height = 70;
10          weight = 195;
11          bmi = weight / (height * height) * 703;
12
13          // print results
14          System.out.println("Previous BMI:");
15          System.out.println(bmi);
16
17          // recompute BMI
18          weight = 180;
19          bmi = weight / (height * height) * 703;
20
21          // report new results
22          System.out.println("Current BMI:");
23          System.out.println(bmi);
24      }
25  }
```

The program begins the same way, setting the three variables to the following values and reporting this initial value for BMI:

But the new program then includes the following assignment statement:

```
weight = 180;
```

This changes the value of the `weight` variable:

You might think that this would also change the value of the `bmi` variable. After all, earlier in the program we said that the following should be true:

```
bmi = weight / (height * height) * 703;
```

This is a place where the spreadsheet analogy is not as accurate. A spreadsheet can store formulas in its cells and when you update one cell it can cause the values in other cells to be updated. The same is not true in Java.

You might also be misled by the use of an equals sign for assignment. Don't confuse this statement with a statement of equality. The assignment statement does not represent an algebraic relationship. In algebra, you might say

$$x = y + 2$$

In mathematics you state definitively that x is equal to y plus two, a fact that is true now and forever. If x changes, y will change accordingly, and vice versa. Java's assignment statement is very different.

The assignment statement is a command to perform an action at a particular point in time. It does not represent a lasting relationship between variables. That's why we usually say "gets" or "is assigned" rather than saying "equals" when we read assignment statements.

Getting back to the program, resetting the variable called `weight` does not reset the variable called `bmi`. To recompute `bmi` based on the new value for `weight`, we must include the second assignment statement:

```
weight = 180;
bmi = weight / (height * height) * 703;
```

Otherwise, the variable `bmi` would store the same value as before. That would be a rather depressing outcome to report to someone who's just lost 15 pounds. By including both of these statements, we reset both the `weight` and `bmi` variables so that memory looks like this:

The output of the new version of the program is

```
Previous BMI:
27.976530612244897
Current BMI:
25.82448979591837
```

One very common assignment statement that points out the difference between algebraic relationships and program statements is:

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

Remember not to think of this as "x equals x + 1." There are no numbers that satisfy that equation. We use a word like "gets" to read this as "x gets the value of x plus one." This may seem a rather odd statement, but you should be able to decipher it given the rules outlined earlier. Suppose that the current value of x is 19. To execute the statement, you first evaluate the expression to obtain the result 20. The computer stores this value in the variable named on the left, x. Thus, this statement adds one to the value of the variable. We refer to this as *incrementing* the value of x. It is a fundamental programming operation because it is the programming equivalent of counting (1, 2, 3, 4, and so on). The following statement is a variation that counts down, which we call *decrementing* a variable:

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

We will discuss incrementing and decrementing in more detail later in this chapter.

Assignment/Declaration Variations

Java is a complex language that provides a lot of flexibility to programmers. In the last section we saw the simplest form of variable declaration and assignment, but there are many variations on this theme. It wouldn't be a bad idea to stick with the simplest form while you are learning, but you'll come across other forms as you read other people's programs, so you'll want to understand what they mean.

The first variation is that Java allows you to provide an initial value for a variable at the time that you declare it. The syntax is as follows:

```
<type> <name> = <expression>;
```

as in

```
double height = 70;
double weight = 195;
double bmi = weight / (height * height) * 703;
```

This variation combines declaration and assignment in one line of code. The first two assignments have simple numbers after the equals sign, but the third has a complex

expression after the equals sign. These three assignments have the same effect as providing three declarations followed by three assignment statements:

```
double height;
double weight;
double bmi;
height = 70;
weight = 195;
bmi = weight / (height * height) * 703;
```

Another variation is to declare several variables that are all of the same type in a single statement. The syntax is

```
<type> <name>, <name>, <name>, ..., <name>;
```

as in

```
double height, weight;
```

This example declares two different variables, both of type `double`. Notice that the type appears just once, at the beginning of the declaration.

The final variation is a mixture of the previous two forms. You can declare multiple variables all of the same type, and you can initialize them at the same time. For example, you could say

```
double height = 70, weight = 195;
```

This statement declares the two `double` variables `height` and `weight` and gives them initial values (`70` and `195`, respectively). Java even allows you to mix initializing and not initializing, as in

```
double height = 70, weight = 195, bmi;
```

This statement declares three `double` variables called `height`, `weight`, and `bmi` and provides initial values to two of them (`height` and `weight`). The variable `bmi` is uninitialized.

Common Programming Error

Declaring the Same Variable Twice

One of the things to keep in mind as you learn is that you can declare any given variable just once. You can assign it as many times as you like once you've declared it, but the declaration should appear just once. Think of variable declaration as being like checking into a hotel and assignment as being like going in

Continued on next page

Continued from previous page

and out of your room. You have to check in first to get your room key, but then you can come and go as often as you like. If you tried to check in a second time, the hotel would be likely to ask you if you really want to pay for a second room.

If you declare a variable more than once, Java generates a compiler error. For example, say your program contains the following lines:

```
int x = 13;
System.out.println(x);
int x = 2;          // this line does not compile
System.out.println(x);
```

The first line is okay. It declares an integer variable called `x` and initializes it to `13`. The second line is also okay, because it simply prints the value of `x`. But the third line will generate an error message indicating that "`x` is already defined." If you want to change the value of `x` you need to use a simple assignment statement instead of a variable declaration:

```
int x = 13;
System.out.println(x);
x = 2;
System.out.println(x);
```

We have been referring to the "assignment statement," but in fact assignment is an operator, not a statement. When you assign a value to a variable, the overall expression evaluates to the value just assigned. That means that you can form expressions that have assignment operators embedded within them. Unlike most other operators, the assignment operator evaluates from right to left, which allows programmers to write statements like the following:

```
int x, y, z;
x = y = z = 2 * 5 + 4;
```

Because the assignment operator evaluates from right to left, this statement is equivalent to:

```
x = (y = (z = 2 * 5 + 4));
```

The expression `2 * 5 + 4` evaluates to `14`. This value is assigned to `z`. The assignment is itself an expression that evaluates to `14`, which is then assigned to `y`. The assignment to `y` evaluates to `14` as well, which is then assigned to `x`. The result is that all three variables are assigned the value `14`.

String Concatenation

You saw in Chapter 1 that you can output string literals using `System.out.println`. You can also output numeric expressions using `System.out.println`:

```
System.out.println(12 + 3 - 1);
```

This statement causes the computer first to evaluate the expression, which yields the value `14`, and then to write that value to the console window. You'll often want to output more than one value on a line, but unfortunately, you can pass only one value to `println`. To get around this limitation, Java provides a simple mechanism called *concatenation* for putting together several pieces into one long string literal.

> **String Concatenation**
>
> Combining several strings into a single string, or combining a string with other data into a new, longer string.

The addition (+) operator concatenates the pieces together. Doing so forms an expression that can be evaluated. Even if the expression includes both numbers and text, it can be evaluated just like the numeric expressions we have been exploring. Consider, for example, the following:

```
"I have " + 3 + " things to concatenate"
```

You have to pay close attention to the quotation marks in an expression like this to keep track of which parts are "inside" a string literal and which are outside. This expression begins with the text `"I have "` (including a space at the end), followed by a plus sign and the integer literal `3`. Java converts the integer into a textual form (`"3"`) and concatenates the two pieces together to form `"I have 3"`. Following the `3` is another plus and another string literal, `" things to concatenate"` (which starts with a space). This piece is glued onto the end of the previous string to form the string `"I have 3 things to concatenate"`.

Because this expression produces a single concatenated string, we can include it in a `println` statement:

```
System.out.println("I have " + 3 + " things to concatenate");
```

This statement produces a single line of output:

```
I have 3 things to concatenate
```

String concatenation is often used to report the value of a variable. Consider, for example, the following program that computes the number of hours, minutes, and seconds in a standard year:

```
1  public class Time {
2      public static void main(String[] args) {
3          int hours = 365 * 24;
```

```
 4          int minutes = hours * 60;
 5          int seconds = minutes * 60;
 6          System.out.println("Hours in a year = " + hours);
 7          System.out.println("Minutes in a year = " + minutes);
 8          System.out.println("Seconds in a year = " + seconds);
 9      }
10  }
```

Notice that the three `println` commands at the end each have a string literal concatenated with a variable. The program produces the following output:

```
Hours in a year = 8760
Minutes in a year = 525600
Seconds in a year = 31536000
```

You can use concatenation to form arbitrarily complex expressions. For example, if you had variables `x`, `y`, and `z` and you wanted to write out their values in coordinate format with parentheses and commas, you could say:

```
System.out.println("(" + x + ", " + y + ", " + z + ")");
```

If `x`, `y`, and `z` had the values `8`, `19`, and `23`, respectively, this statement would output the string `"(8, 19, 23)"`.

The + used for concatenation has the same level of precedence as the normal arithmetic + operator, which can lead to some confusion. Consider, for example, the following expression:

```
2 + 3 + " hello " + 7 + 2 * 3
```

This expression has four addition operators and one multiplication operator. Because of precedence, we evaluate the multiplication first:

```
2 + 3 + " hello " + 7 + 2 * 3
2 + 3 + " hello " + 7 +     6
```

This grouping might seem odd, but that's what the precedence rule says to do: We don't evaluate any additive operators until we've first evaluated all of the multiplicative operators. Once we've taken care of the multiplication, we're left with the four addition operators. These will be evaluated from left to right.

The first addition involves two integer values. Even though the overall expression involves a string, because this little subexpression has just two integers we perform integer addition:

```
2 + 3 + " hello " + 7 + 6
  5   + " hello " + 7 + 6
```

The next addition involves adding the integer 5 to the string literal " hello ". If either of the two operands is a string, we perform concatenation. So, in this case, we convert the integer into a text equivalent ("5") and glue the pieces together to form a new string value:

```
5  +  " hello "  +  7  +  6
   "5 hello "     +  7  +  6
```

You might think that Java would add together the 7 and 6 the same way it added the 2 and 3 to make 5. But it doesn't work that way. The rules of precedence are simple, and Java follows them with simple-minded consistency. Precedence tells us that addition operators are evaluated from left to right, so first we add the string "5 hello " to 7. That is another combination of a string and an integer, so Java converts the integer to its textual equivalent ("7") and concatenates the two parts together to form a new string:

```
"5 hello "  +  7  +  6
  "5 hello 7"     +  6
```

Now there is just a single remaining addition to perform, which again involves a string/integer combination. We convert the integer to its textual equivalent ("6") and concatenate the two parts together to form a new string:

```
"5 hello 7"  +  6
   "5 hello 76"
```

Clearly, such expressions can be confusing, but you wouldn't want the Java compiler to have to try to guess what you mean. Our job as programmers is easier if we know that the compiler is going to follow simple rules consistently. You can make the expression clearer, and specify how it is evaluated, by adding parentheses. For example, if we really did want Java to add together the 7 and 6 instead of concatenating them separately, we could have written the original expression in the following much clearer way:

```
(2 + 3) + " hello " + (7 + 2 * 3)
```

Because of the parentheses, Java will evaluate the two numeric parts of this expression first and then concatenate the results with the string in the middle. This expression evaluates to "5 hello 13".

Increment/Decrement Operators

In addition to the standard assignment operator, Java has several special operators that are useful for a particular family of operations that are common in programming. As we mentioned earlier, you will often find yourself increasing the value of a variable by a particular amount, an operation called *incrementing*. You will also often

find yourself decreasing the value of a variable by a particular amount, an operation called *decrementing*. To accomplish this, you write statements like the following:

```
x = x + 1;
y = y - 1;
z = z + 2;
```

Likewise, you'll frequently find yourself wanting to double or triple the value of a variable or to reduce its value by a factor of 2, in which case you might write code like the following:

```
x = x * 2;
y = y * 3;
z = z / 2;
```

Java has a shorthand for these situations. You glue together the operator character (+, –, *, etc.) with the equals sign to get a special assignment operator (+=, –=, *=, etc.). This variation allows you to rewrite assignment statements like the previous ones as follows:

```
x += 1;
y -= 1;
z += 2;

x *= 2;
y *= 3;
z /= 2;
```

This convention is yet another detail to learn about Java, but it can make the code easier to read. Think of a statement like `x += 2` as saying, "add `2` to `x`." That's more concise than saying `x = x + 2`.

Java has an even more concise way of expressing the particular case in which you want to increment by 1 or decrement by 1. In this case, you can use the increment and decrement operators (++ and ––). For example, you can say

```
x++;
y--;
```

There are actually two different forms of each of these operators, because you can also put the operator in front of the variable:

```
++x;
--y;
```

The two versions of ++ are known as the preincrement (`++x`) and postincrement (`x++`) operators. The two versions of –– are similarly known as the predecrement

Table 2.5 Java Operator Precedence

Description	Operators
unary operators	`++, --, +, -`
multiplicative operators	`*, /, %`
additive operators	`+, -`
assignment operators	`=, +=, -=, *=, /=, %=`

(--x) and postdecrement (x--) operators. The pre- versus post- distinction doesn't matter when you include them as statements by themselves, as in these two examples. The difference comes up only when you embed these statements inside more complex expressions, which we don't recommend.

Now that we've seen a number of new operators, it is worth revisiting the issue of precedence. Table 2.5 shows an updated version of the Java operator precedence table that includes the assignment operators and the increment and decrement operators. Notice that the increment and decrement operators are grouped with the unary operators and have the highest precedence.

Did You Know?

++ and --

The ++ and -- operators were first introduced in the C programming language. Java has them because the designers of the language decided to use the syntax of C as the basis for Java syntax. Many languages have made the same choice, including C++ and C#. There is almost a sense of pride among C programmers that these operators allow you to write extremely concise code, but many other people feel that they can make code unnecessarily complex. In this book we always use these operators as separate statements so that it is obvious what is going on, but in the interest of completeness we will look at the other option here.

The pre- and post- variations both have the same overall effect—the two increment operators increment a variable and the two decrement operators decrement a variable—but they differ in terms of what they evaluate to. When you increment or decrement, there are really two values involved: the original value that the variable had before the increment or decrement operation, and the final value that the variable has after the increment or decrement operation. The post- versions evaluate to the original (older) value and the pre- versions evaluate to the final (later) value.

Consider, for example, the following code fragment:

```
int x = 10;
int y = 20;
int z = ++x * y--;
```

Continued on next page

Continued from previous page

What value is `z` assigned? The answer is `220`. The third assignment increments `x` to 11 and decrements `y` to 19, but in computing the value of `z`, it uses the new value of `x` (`++x`) times the old value of `y` (`y--`), which is 11 times 20, or 220.

There is a simple mnemonic to remember this: When you see `x++`, read it as "give me `x`, then increment," and when you see `++x`, read it as "increment, then give me `x`." Another memory device that might help is to remember that C++ is a bad name for a programming language. The expression "C++" would be interpreted as "evaluate to the old value of C and then increment C." In other words, even though you're trying to come up with something new and different, you're really stuck with the old awful language. The language you want is ++C, which would be a new and improved language rather than the old one. Some people have suggested that perhaps Java is ++C.

Variables and Mixing Types

You already know that when you declare a variable, you must tell Java what type of value it will be storing. For example, you might declare a variable of type `int` for integer values or of type `double` for real values. The situation is fairly clear when you have just integers or just reals, but what happens when you start mixing the types? For example, the following code is clearly okay:

```
int x;
double y;
x = 2 + 3;
y = 3.4 * 2.9;
```

Here, we have an integer variable that we assign an integer value and a `double` variable that we assign a `double` value. But what if we try to do it the other way around?

```
int x;
double y;
x = 3.4 * 2.9; // illegal
y = 2 + 3;     // okay
```

As the comments indicate, you can't assign an integer variable a `double` value, but you can assign a `double` variable an integer value. Let's consider the second case first. The expression `2 + 3` evaluates to the integer `5`. This value isn't a `double`, but every integer is a real value, so it is easy enough for Java to convert the integer into a `double`. The technical term is that Java *promotes* the integer into a `double`.

The first case is more problematic. The expression `3.4 * 2.9` evaluates to the `double` value `9.86`. This value can't be stored in an integer because it isn't an integer. If you want to perform this kind of operation, you'll have to tell Java to convert this

value into an integer. As described earlier, you can cast a `double` to an `int`, which will truncate anything after the decimal point:

```
x = (int) (3.4 * 2.9);  // now legal
```

This statement first evaluates `3.4 * 2.9` to get `9.86` and then truncates that value to get the integer `9`.

Common Programming Error

Forgetting to Cast

We often write programs that involve a mixture of `int`s and `double`s, so it is easy to make mistakes when it comes to combinations of the two. For example, suppose that you want to compute the percentage of correctly answered questions on a student's test, given the total number of questions on the test and the number of questions the student got right. You might declare the following variables:

```
int totalQuestions;
int numRight;
double percent;
```

Suppose the first two are initialized as follows:

```
totalQuestions = 73;
numRight = 59;
```

How do you compute the percentage of questions that the student got right? You divide the number right by the total number of questions and multiply by 100 to turn it into a percentage:

```
percent = numRight / totalQuestions * 100; // incorrect
```

Unfortunately, if you print out the value of the variable `percent` after executing this line of code, you will find that it has the value `0.0`. But obviously the student got more than 0% correct.

The problem comes from integer division. The expression you are using begins with two `int` values:

```
numRight / totalQuestions
```

which means you are computing

```
59 / 73
```

Continued on next page

Continued from previous page

This evaluates to 0 with integer division. Some students fix this by changing the types of all the variables to double. That will solve the immediate problem, but it's not a good choice from a stylistic point of view. It is best to use the most appropriate type for data, and the number of questions on the test will definitely be an integer. You could try to fix this by changing the value 100 to 100.0:

```
percent = numRight / totalQuestions * 100.0; // incorrect
```

but this doesn't help because the division is done first. However, it does work if you put the 100.0 first:

```
percent = 100.0 * numRight / totalQuestions;
```

Now the multiplication is computed before the division, which means that everything is converted to double.

Sometimes you can fix a problem like this through a clever rearrangement of the formula, but you don't want to count on cleverness. This is a good place to use a cast. For example, returning to the original formula, you can cast each of the int variables to double:

```
percent = (double) numRight / (double) totalQuestions * 100.0;
```

You can also take advantage of the fact that once you have cast one of these two variables to double, the division will be done with doubles. So you could, for example, cast just the first value to double:

```
percent = (double) numRight / totalQuestions * 100.0;
```

2.3 The for Loop

Programming often involves specifying redundant tasks. The for loop helps to avoid such redundancy by repeatedly executing a sequence of statements over a particular range of values. Suppose you want to write out the squares of the first five integers. You could write a program like this:

```
1  public class WriteSquares {
2      public static void main(String[] args) {
3          System.out.println(1 + " squared = " + (1 * 1));
4          System.out.println(2 + " squared = " + (2 * 2));
5          System.out.println(3 + " squared = " + (3 * 3));
6          System.out.println(4 + " squared = " + (4 * 4));
7          System.out.println(5 + " squared = " + (5 * 5));
8      }
9  }
```

which would produce the following output:

```
1 squared = 1
2 squared = 4
3 squared = 9
4 squared = 16
5 squared = 25
```

But this approach is tedious. The program has five statements that are very similar. They are all of the form:

```
System.out.println(number + " squared = " + (number * number));
```

where `number` is either `1`, `2`, `3`, `4`, or `5`. The `for` loop avoids such redundancy. Here is an equivalent program using a `for` loop:

```
1  public class WriteSquares2 {
2      public static void main(String[] args) {
3          for (int i = 1; i <= 5; i++) {
4              System.out.println(i + " squared = " + (i * i));
5          }
6      }
7  }
```

This program initializes a variable called `i` to the value `1`. Then it repeatedly executes the `println` statement as long as the variable `i` is less than or equal to `5`. After each `println`, it evaluates the expression `i++` to increment `i`.

The general syntax of the `for` loop is as follows:

```
for (<initialization>; <continuation test>; <update>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

You always include the keyword `for` and the parentheses. Inside the parentheses are three different parts, separated by semicolons: the initialization, the continuation test, and the update. Then there is a set of curly braces that encloses a set of statements. The `for` loop controls the statements inside the curly braces. We refer to the controlled statements as the *body* of the loop. The idea is that we execute the body multiple times, as determined by the combination of the other three parts.

The diagram in Figure 2.1 indicates the steps that Java follows to execute a `for` loop. It performs whatever initialization you have requested once before the loop begins executing. Then it repeatedly performs the continuation test you have provided.

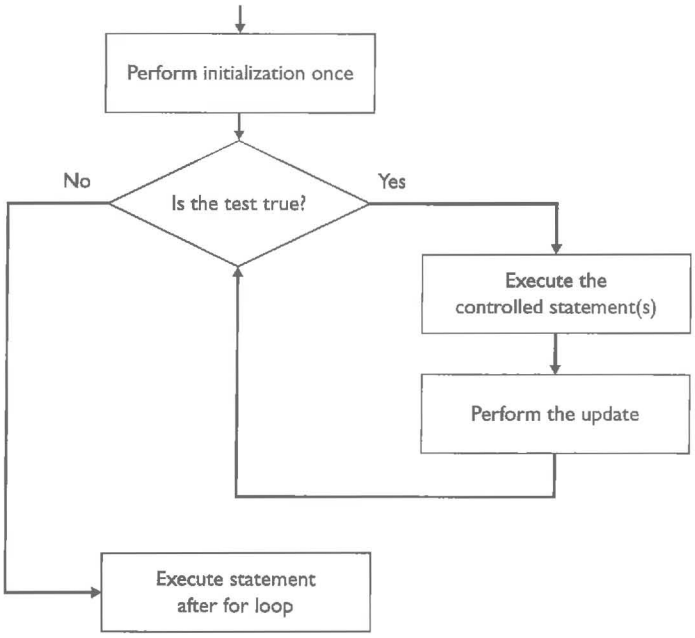


Figure 2.1 Flow of `for` loop

If the continuation test evaluates to `true`, it executes the controlled statements once and executes the update part. Then it performs the test again. If it again evaluates to `true`, it executes the statements again and executes the update again. Notice that the update is performed after the controlled statements are executed. When the test evaluates to `false`, Java is done executing the loop and moves on to whatever statement comes after the loop.

The `for` loop is the first example of a *control structure* that we will study.

> **Control Structure**
>
> A syntactic structure that controls other statements.

You should be careful to use indentation to indicate controlled statements. In the case of the `for` loop, all of the statements in the body of the loop are indented as a way to indicate that they are "inside" the loop.

Tracing `for` Loops

Let's examine the `for` loop of the `WriteSquares2` program in detail:

```
for (int i = 1; i <= 5; i++) {
    System.out.println(i + " squared = " + (i * i));
}
```

Table 2.6 Trace of `for (int i = 1; i <= 5; i++)`

Step	Code	Description
initialization	`int i = 1;`	variable `i` is created and initialized to `1`
test	`i <= 5`	`true` because `1 <= 5`, so we enter the loop
body	`{ . . . }`	execute the `println` with `i` equal to `1`
update	`i++`	increment `i`, which becomes `2`
test	`i <= 5`	`true` because `2 <= 5`, so we enter the loop
body	`{ . . . }`	execute the `println` with `i` equal to `2`
update	`i++`	increment `i`, which becomes `3`
test	`i <= 5`	`true` because `3 <= 5`, so we enter the loop
body	`{ . . . }`	execute the `println` with `i` equal to `3`
update	`i++`	increment `i`, which becomes `4`
test	`i <= 5`	`true` because `4 <= 5`, so we enter the loop
body	`{ . . . }`	execute the `println` with `i` equal to `4`
update	`i++`	increment `i`, which becomes `5`
test	`i <= 5`	`true` because `5 <= 5`, so we enter the loop
body	`{ . . . }`	execute the `println` with `i` equal to `5`
update	`i++`	increment `i`, which becomes `6`
test	`i <= 5`	`false` because `6 > 5`, so we are finished

In this loop, the initialization (`int i = 1`) declares an integer variable `i` that is initialized to `1`. The continuation test (`i <= 5`) indicates that we should keep executing as long as `i` is less than or equal to `5`. That means that once `i` is greater than `5`, we will stop executing the body of the loop. The update (`i++`) will increment the value of `i` by one each time, bringing `i` closer to being larger than `5`. After five executions of the body and the accompanying five updates, `i` will be larger than 5 and the loop will finish executing. Table 2.6 traces this process in detail.

Java allows great flexibility in deciding what to include in the initialization part and the update, so we can use the `for` loop to solve all sorts of programming tasks. For now, though, we will restrict ourselves to a particular kind of loop that declares and initializes a single variable that is used to control the loop. This variable is often referred to as the *control variable* of the loop. In the test we compare the control variable against some final desired value, and in the update we change the value of the control variable, most often incrementing it by `1`. Such loops are very common in programming. By convention, we often use names like `i`, `j`, and `k` for the control variables.

Each execution of the controlled statement of a loop is called an *iteration* of the loop (as in, "The loop finished executing after four iterations"). Iteration also refers to looping in general (as in, "I solved the problem using iteration").

Consider another `for` loop:

```
for (int i = -100; i <= 100; i++) {
    System.out.println(i + " squared = " + (i * i));
}
```

This loop executes a total of 201 times, producing the squares of all the integers between –100 and +100 inclusive. The values used in the initialization and the test, then, can be any integers. They can, in fact, be arbitrary integer expressions:

```
for (int i = (2 + 2); i <= (17 * 3); i++) {
    System.out.println(i + " squared = " + (i * i));
}
```

This loop will generate the squares of all the integers between 4 and 51 inclusive. The parentheses around the expressions are not necessary but improve readability. Consider the following loop:

```
for (int i = 1; i <= 30; i++) {
    System.out.println("+--------+");
}
```

This loop generates 30 lines of output, all exactly the same. It is slightly different from the previous one because the statement controlled by the for loop makes no reference to the control variable. Thus,

```
for (int i = –30; i <= –1; i++) {
    System.out.println("+--------+");
}
```

generates exactly the same output. The behavior of such a loop is determined solely by the number of iterations it performs. The number of iterations is given by

```
<ending value> – <starting value> + 1
```

It is much simpler to see that the first of these loops iterates 30 times, so it is better to use that loop.

Now let's look at some borderline cases. Consider this loop:

```
for (int i = 1; i <= 1; i++) {
    System.out.println("+--------+");
}
```

According to our rule it should iterate once, and it does. It initializes the variable i to 1 and tests to see if this is less than or equal to 1, which it is. So it executes the println, increments i, and tests again. The second time it tests, it finds that i is no longer less than or equal to 1, so it stops executing. Now consider this loop:

```
for (int i = 1; i <= 0; i++) {
    System.out.println("+--------+"); // never executes
}
```

This loop performs no iterations at all. It will not cause an execution error; it just won't execute the body. It initializes the variable to `1` and tests to see if this is less than or equal to `0`. It isn't, so rather than executing the statements in the body, it stops there.

When you construct a `for` loop, you can include more than one statement inside the curly braces. Consider, for example, the following code:

```
for (int i = 1; i <= 20; i++) {
     System.out.println("Hi!");
     System.out.println("Ho!");
}
```

This will produce 20 pairs of lines, the first of which has the word "Hi!" on it and the second of which has the word "Ho!"

When a `for` loop controls a single statement, you don't have to include the curly braces. The curly braces are required only for situations like the previous one, where you have more than one statement that you want the loop to control. However, the Java coding convention includes the curly braces even for a single statement, and we follow this convention in this book. There are two advantages to this convention:

- Including the curly braces prevents future errors. Even if you need only one statement in the body of your loop now, your code is likely to change over time. Having the curly braces there ensures that, if you add an extra statement to the body later, you won't accidentally forget to include them. In general, including curly braces in advance is cheaper than locating obscure bugs later.
- Always including the curly braces reduces the level of detail you have to consider as you learn new control structures. It takes time to master the details of any new control structure, and it will be easier to master those details if you don't have to also be thinking about when to include and when not to include the braces.

Common Programming Error

Forgetting Curly Braces

You should use indentation to indicate the body of a `for` loop, but indentation alone is not enough. Java ignores indentation when it is deciding how different statements are grouped. Suppose, for example, that you were to write the following code:

```
for (int i = 1; i <= 20; i++)
    System.out.println("Hi!");
    System.out.println("Ho!");
```

The indentation indicates to the reader that both of the `println` statements are in the body of the `for` loop, but there aren't any curly braces to indicate that to Java. As a result, this code is interpreted as follows:

Continued on next page

Continued from previous page

```
for (int i = 1; i <= 20; i++) {
    System.out.println("Hi!");
}
System.out.println("Ho!");
```

Only the first `println` is considered to be in the body of the `for` loop. The second `println` is considered to be outside the loop. So, this code would produce 20 lines of output that all say "Hi!" followed by one line of output that says "Ho!" To include both `println`s in the body, you need curly braces around them:

```
for (int i = 1; i <= 20; i++) {
    System.out.println("Hi!");
    System.out.println("Ho!");
}
```

for Loop Patterns

In general, if you want a loop to iterate exactly *n* times, you will use one of two standard loops. The first standard form looks like the ones you have already seen:

```
for (int <variable> = 1; <variable> <= n; i++) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

It's pretty clear that this loop executes *n* times, because it starts at `1` and continues as long as it is less than or equal to *n*. For example, this loop prints the numbers 1 through 10:

```
for (int i = 1; i <= 10; i++) {
    System.out.print(i + " ");
}
```

Because it uses a `print` instead of a `println` statement, it produces a single line of output:

```
1 2 3 4 5 6 7 8 9 10
```

Often, however, it is more convenient to start our counting at `0` instead of `1`. That requires a change in the loop test to allow you to stop when *n* is one less:

```
for (int <variable> = 0; <variable> < n; i++) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

Notice that in this form when you initialize the variable to `0`, you test whether it is strictly less than *n*. Either form will execute exactly *n* times, although there are some situations where the zero-based loop works better. For example, this loop executes 10 times just like the previous loop:

```
for (int i = 0; i < 10; i++) {
    System.out.print(i + " ");
}
```

Because it starts at `0` instead of starting at `1`, it produces a different sequence of 10 values:

```
0 1 2 3 4 5 6 7 8 9
```

Most often you will use the loop that starts at `0` or `1` to perform some operation a fixed number of times. But there is a slight variation that is also sometimes useful. Instead of running the loop in a forward direction, we can run it backward. Instead of starting at `1` and executing until you reach *n*, you instead start at *n* and keep executing until you reach `1`. You can accomplish this by using a decrement rather than an increment, so we sometimes refer to this as a decrementing loop.

Here is the general form of a decrementing loop:

```
for (int <variable> = n; <variable> >= 1; <variable>--) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

For example, here is a decrementing loop that executes 10 times:

```
for (int i = 10; i >= 1; i--) {
    System.out.print(i + " ");
}
```

Because it runs backward, it prints the values in reverse order:

```
10 9 8 7 6 5 4 3 2 1
```

Nested for Loops

The `for` loop controls a statement, and the `for` loop is itself a statement, which means that one `for` loop can control another `for` loop. For example, you can write code like the following:

```
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 5; j++) {
        System.out.println("Hi there.");
    }
}
```

This code is probably easier to read from the inside out. The `println` statement produces a single line of output. The inner `j` loop executes this statement five times, producing five lines of output. The outer `i` loop executes the inner loop 10 times, which produces 10 sets of 5 lines, or 50 lines of output. The preceding code, then, is equivalent to

```
for (int i = 1; i <= 50; i++) {
    System.out.println("Hi there.");
}
```

This example shows that a `for` loop can be controlled by another `for` loop. Such a loop is called a *nested loop*. This example wasn't very interesting, though, because the nested loop can be eliminated.

Now that you know how to write `for` loops, you will want to be able to produce complex lines of output piece by piece using the `print` command. Recall from Chapter 1 that the `print` command prints on the current line of output without going to a new line of output. For example, if you want to produce a line of output that has 80 stars on it, you can use a `print` command to print one star at a time and have it execute 80 times rather than using a single `println`.

Let's look at a more interesting nested loop that uses a `print` command:

```
for (int i = 1; i <= 6; i++) {
    for (int j = 1; j <= 3; j++) {
        System.out.print(j + " ");
    }
}
```

We can once again read this from the inside out. The inner loop prints the value of its control variable `j` as it varies from `1` to `3`. The outer loop executes this six different times. As a result, we get six occurrences of the sequence `1 2 3` as output:

```
1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

This code prints all of its output on a single line of output. Let's look at some code that includes a combination of `print` and `println` to produce several lines of output:

```
for (int i = 1; i <= 6; i++) {
    for (int j = 1; j <= 10; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

When you write code that involves nested loops, you have to be careful to indent the code correctly to make the structure clear. At the outermost level, the preceding code is a simple `for` loop that executes six times:

```
for (int i = 1; i <= 6; i++) {
    ...
}
```

We use indentation for the statements inside this `for` loop to make it clear that they are the body of this loop. Inside, we find two statements: another `for` loop and a `println`. Let's look at the inner `for` loop:

```
for (int j = 1; j <= 10; j++) {
    System.out.print("*");
}
```

This loop is controlled by the outer `for` loop, which is why it is indented, but it itself controls a statement (the `print` statement), so we end up with another level of indentation. The indentation thus indicates that the `print` statement is controlled by the inner `for` loop, which in turn is controlled by the outer `for` loop. So what does this inner loop do? It prints 10 stars on the current line of output. They all appear on the same line because we are using a `print` instead of a `println`. Notice that after this loop we perform a `println`:

```
System.out.println();
```

The net effect of the `for` loop followed by the `println` is that we get a line of output with 10 stars on it. But remember that these statements are contained in an outer loop that executes six times, so we end up getting six lines of output, each with 10 stars:

```
**********
**********
**********
**********
**********
**********
```

Let's examine one more variation. In the code above, the inner `for` loop always does exactly the same thing: It prints exactly 10 stars on a line of output. But what happens if we change the test for the inner `for` loop to make use of the outer `for` loop's control variable (`i`)?

```
for (int i = 1; i <= 6; i++) {
    for (int j = 1; j <= i; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

In the old version the inner loop always executes 10 times, producing 10 stars on each line of output. With the new test (`j <= i`), the inner loop will execute `i` times with each iteration. But `i` is changing: It takes on the values `1`, `2`, `3`, `4`, `5`, and `6`. On the first iteration of the outer loop, when `i` is `1`, the test `j <= i` is effectively testing `j <= 1`, and it generates a line with one star on it. On the second iteration of the outer loop, when `i` is `2`, the test is effectively testing `j <= 2`, and it generates a line with two stars on it. On the third iteration of the outer loop, when `i` is `3`, the test is effectively testing `j <= 3`, and it generates a line with three stars on it. This continues through the sixth iteration.

In other words, this code produces a triangle as output:

```
*
**
***
****
*****
******
```

2.4 Managing Complexity

You've learned about several new programming constructs in this chapter, and it's time to put the pieces together to solve some complex tasks. As we pointed out in Chapter 1, Brian Kernighan, one of the coauthors of *The C Programming Language*, has said that "Controlling complexity is the essence of computer programming." In this section we will examine several techniques that computer scientists use to solve complex problems without being overwhelmed by complexity.

Scope

As programs get longer, it is increasingly likely that different parts of the program will interfere with each other. Java helps us to manage this potential problem by enforcing rules of *scope*.

Scope

The part of a program in which a particular declaration is valid.

As you've seen, when it comes to declaring static methods, you can put them in any order whatsoever. The scope of a static method is the entire class in which it appears. Variables work differently. The simple rule is that the scope of a variable declaration extends from the point where it is declared to the right curly brace that encloses it. In other words, find the pair of curly braces that directly encloses the variable declaration. The scope of the variable is from the point where it is declared to the closing curly brace.

This scope rule has several implications. Consider first what it means for different methods. Each method has its own set of curly braces to indicate the statements to be executed when the method is called. Any variables declared inside a method's curly braces won't be available outside the method. We refer to such variables as *local variables,* and we refer to the process of limiting their scope as *localizing* variables.

Local Variable

A variable declared inside a method that is accessible only in that method.

Localizing Variables

Declaring variables in the innermost (most local) scope possible.

In general, you will want to declare variables in the most local scope possible. You might wonder why we would want to localize variables to just one method. Why not just declare everything in one outer scope? That certainly seems simpler, but there are some important drawbacks. Localizing variables leads to some duplication (and possibly confusion) but provides more security. As an analogy, consider the use of refrigerators in dormitories. Every dorm room can have its own refrigerator, but if you are outside a room, you don't know whether it has a refrigerator in it. The contents of the room are hidden from you.

Java programs use variables to store values just as students use refrigerators to store beer, ice cream, and other valuables. The last time we were in a dorm we noticed that most of the individual rooms had refrigerators in them. This seems terribly redundant, but the reason is obvious. If you want to guarantee the security of something, you put it where nobody else can get it. You will use local variables in your programs in much the same way. If each individual method has its own local variables to use, you don't have to consider possible interference from other parts of the program.

Let's look at a simple example involving two methods:

```
1  // This program does not compile.
2  public class ScopeExample {
3        public static void main(String[] args) {
```

```
4           int x = 3;
5           int y = 7;
6           computeSum();
7       }
8
9       public static void computeSum() {
10          int sum = x + y; // illegal, x/y are not in scope
11          System.out.println("sum = " + sum);
12      }
13  }
```

In this example, the `main` method declares local variables `x` and `y` and gives them initial values. Then it calls the method `computeSum`. Inside this method, we try to use the values of `x` and `y` to compute a sum. However, because the variables `x` and `y` are local to the `main` method and are not visible inside of the `computeSum` method, this doesn't work. (In the next chapter, we will see a technique for allowing one method to pass a value to another.)

The program produces error messages like the following:

```
ScopeExample.java:10: error: cannot find symbol
symbol   : variable x
location: class ScopeExample
        int sum = x + y;  // illegal, x/y are not in scope
                  ^
ScopeExample.java:10: error: cannot find symbol
symbol   : variable y
location: class ScopeExample
         int sum = x + y;  // illegal, x/y are not in scope
                       ^
```

It's important to understand scope in discussing the local variables of one method versus another. Scope also has implications for what happens inside a single method. You have seen that curly braces are used to group together a series of statements. But you can have curly braces inside curly braces, and this leads to some scope issues. For example, consider the following code:

```
for (int i = 1; i <= 5; i++) {
    int squared = i * i;
    System.out.println(i + " squared = " + squared);
}
```

This is a variation of the code we looked at earlier in the chapter to print out the squares of the first five integers. In this version, a variable called `squared` is used to

keep track of the square of the `for` loop control variable. This code works fine, but consider this variation:

```
for (int i = 1; i <= 5; i++) {
    int squared = i * i;
    System.out.println(i + " squared = " + squared);
}
System.out.println("Last square = " + squared); // illegal
```

This code generates a compiler error. The variable `squared` is declared inside the `for` loop. In other words, the curly braces that contain it are the curly braces for the loop. It can't be used outside this scope, so when you attempt to refer to it outside the loop, you'll get a compiler error.

If for some reason you need to write code like this that accesses the variable after the loop, you have to declare the variable in the outer scope before the loop:

```
int squared = 0;  // declaration is now in outer scope
for (int i = 1; i <= 5; i++) {
    squared = i * i;  // change this to an assignment statement
    System.out.println(i + " squared = " + squared);
}
System.out.println("Last square = " + squared); // now legal
```

There are a few special cases for scope, and the `for` loop is one of them. When a variable is declared in the initialization part of a `for` loop, its scope is just the `for` loop itself (the three parts in the `for` loop header and the statements controlled by the `for` loop). That means you can use the same variable name in multiple `for` loops:

```
for (int i = 1; i <= 10; i++) {
    System.out.println(i + " squared = " + (i * i));
}
for (int i = 1; i <= 10; i++) {
    System.out.println(i + " cubed = " + (i * i * i));
}
```

The variable `i` is declared twice in the preceding code, but because the scope of each variable is just the `for` loop in which it is declared, this isn't a problem. (It's like having two dorm rooms, each with its own refrigerator.) Of course, you can't do this with nested `for` loops. The following code, for example, will not compile:

```
for (int i = 1; i <= 5; i++) {
    for (int i = 1; i <= 10; i++) { // illegal
        System.out.println("hi there.");
    }
}
```

When Java encounters the inner `for` loop, it will complain that the variable `i` has already been declared within this scope. You can't declare the same variable twice within the same scope. You have to come up with two different names to distinguish between them, just as when there are two Carls in the same family they tend to be called "Carl Junior" and "Carl Senior" to avoid any potential confusion.

A control variable that is used in a `for` loop doesn't have to be declared in the initialization part of the loop. You can separate the declaration of the `for` loop control variable from the initialization of the variable, as in the following code:

```
int i;
for (i = 1; i <= 5; i++) {
    System.out.println(i + " squared = " + (i * i));
}
```

Doing so extends the variable's scope to the end of the enclosing set of curly braces. One advantage of this approach is that it enables you to refer to the final value of the control variable after the loop. Normally you wouldn't be able to do this, because the control variable's scope would be limited to the loop itself. However, declaring the control variable outside the loop is a dangerous practice, and it provides a good example of the problems you can encounter when you don't localize variables. Consider the following code, for example:

```
int i;
for (i = 1; i <= 5; i++) {
    for (i = 1; i <= 10; i++) {
        System.out.println("hi there.");
    }
}
```

As noted earlier, you shouldn't use the same control variable when you have nested loops. But unlike the previous example, this code compiles, because here the variable declaration is outside the outer `for` loop. So, instead of getting a helpful error message from the Java compiler, you get a program with a bug in it. You'd think from reading these loops that the code will produce 50 lines of output, but it actually produces just 10 lines of output. The inner loop increments the variable `i` until it becomes `11`, and that causes the outer loop to terminate after just one iteration. It can be even worse if you reverse the order of these loops:

```
int i;
for (i = 1; i <= 10; i++) {
    for (i = 1; i <= 5; i++) {
        System.out.println("hi there.");
    }
}
```

This code has an *infinite loop.*

Infinite Loop

A loop that never terminates.

This loop is infinite because no matter what the outer loop does to the variable `i`, the inner loop always sets it back to `1` and iterates until it becomes `6`. The outer loop then increments the variable to `7` and finds that `7` is less than or equal to `10`, so it always goes back to the inner loop, which once again sets the variable back to `1` and iterates up to `6`. This process goes on indefinitely. These are the kinds of interference problems you can get when you fail to localize variables.

Common Programming Error

Referring to the Wrong Loop Variable

The following code is intended to print a triangle of stars. However, it has a subtle bug that causes it to print stars infinitely:

```
for (int i = 1; i <= 6; i++) {
    for (int j = 1; j <= i; i++){
        System.out.print("*");
    }
    System.out.println();
}
```

The problem is in the second line, in the inner `for` loop header's update statement. The programmer meant to write `j++` but instead accidentally wrote `i++`. A trace of the code is shown in Table 2.7.

Table 2.7 Trace of Nested `for` Loop

Step	Code	Description
initialization	`int i = 1;`	variable `i` is created and initialized to `1`
initialization	`int j = 1;`	variable `j` is created and initialized to `1`
test	`j <= i`	`true` because `1 <= 1`, so we enter the inner loop
body	`{...}`	execute the `print` with `j` equal to `1`
update	`i++`	increment `i`, which becomes `2`
test	`j <= i`	`true` because `1 <= 2`, so we enter the inner loop
body	`{...}`	execute the `print` with `j` equal to `1`
update	`i++`	increment `i`, which becomes `3`
...	...	...

The variable `j` should be increasing, but instead `i` is increasing. The effect of this mistake is that the variable `j` is never incremented in the inner loop, and therefore the test of `j <= i` never fails, so the inner loop doesn't terminate.

Continued on next page

Continued from previous page

Here's another broken piece of code. This one tries to print a 6×4 box of stars, but it also prints infinitely:

```
for (int i = 1; i <= 6; i++) {
    for (int j = 1; i <= 4; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

The problem is on the second line, this time in the inner `for` loop header's test. The programmer meant to write `j <= 4` but instead accidentally wrote `i <= 4`. Since the value of `i` is never incremented in the inner loop, the test of `i <= 4` never fails, so the inner loop again doesn't terminate.

Pseudocode

As you write more complex algorithms, you will find that you can't just write the entire algorithm immediately. Instead, you will increasingly make use of the technique of writing *pseudocode*.

> **Pseudocode**
>
> English-like descriptions of algorithms. Programming with pseudocode involves successively refining an informal description until it is easily translated into Java.

For example, you can describe the problem of drawing a box as

```
draw a box with 50 lines and 30 columns of asterisks.
```

While this statement describes the figure, it does not give specific instructions about how to draw it (that is, what algorithm to use). Do you draw the figure line by line or column by column? In Java, figures like these must be generated line by line, because once a `println` has been performed on a line of output, that line cannot be changed. There is no command for going back to a previous line in the output. Therefore, you must output the first line in its entirety, then the second line in its entirety, and so on. As a result, your decompositions for figures such as these will be line-oriented at the top level. Thus, a version of the statement that is closer to Java is

```
for (each of 50 lines) {
    draw a line of 30 asterisks.
}
```

This instruction can be made more specific by introducing the idea of repeatedly writing a single character on the output line and then moving to a new line of output:

```
for (each of 50 lines) {
    for (each of 30 columns) {
        write one asterisk on the output line.
    }
    go to a new output line.
}
```

Using pseudocode, you can gradually convert an English description into something that is easily translated into a Java program. The simple examples we've looked at so far are hardly worth the application of pseudocode, so we will now examine the problem of generating a more complex figure:

```
*********
 *******
  *****
   ***
    *
```

This figure must also be generated line by line:

```
for (each of 5 lines) {
    draw one line of the triangle.
}
```

Unfortunately, each line is different. Therefore, you must come up with a general rule that fits all the lines. The first line of this figure has a series of asterisks on it with no leading spaces. Each of the subsequent lines has a series of spaces followed by a series of asterisks. Using your imagination a bit, you can say that the first line has 0 spaces on it followed by a series of asterisks. This allows you to write a general rule for making this figure:

```
for (each of 5 lines) {
    write some spaces (possibly 0) on the output line.
    write some asterisks on the output line.
    go to a new output line.
}
```

In order to proceed, you must determine a rule for the number of spaces and a rule for the number of asterisks. Assuming that the lines are numbered 1 through 5, looking at the figure, you can fill in Table 2.8.

You want to find a relationship between line number and the other two columns. This is simple algebra, because these columns are related in a linear way. The second

Table 2.8 Analysis of Figure

Line	Spaces	Asterisks
1	0	9
2	1	7
3	2	5
4	3	3
5	4	1

column is easy to get from the line number. It equals `(line – 1)`. The third column is a little tougher. Because it goes down by 2 every time and the first column goes up by 1 every time, you need a multiplier of –2. Then you need an appropriate constant. The number 11 seems to do the trick, so you can make the third column equal `(11 – 2 * line)`. You can improve your pseudocode, then, as follows:

```
for (line going 1 to 5) {
    write (line – 1) spaces on the output line.
    write (11 – 2 * line) asterisks on the output line.
    go to a new output line.
}
```

This pseudocode is simple to turn into a program:

```
 1  public class DrawV {
 2      public static void main(String[] args) {
 3          for (int line = 1; line <= 5; line++) {
 4              for (int i = 1; i <= (line – 1); i++) {
 5                  System.out.print(" ");
 6              }
 7              for (int i = 1; i <= (11 – 2 * line); i++) {
 8                  System.out.print("*");
 9              }
10              System.out.println();
11          }
12      }
13  }
```

Sometimes we manage complexity by taking advantage of work that we have already done. For example, how would you produce this figure?

```
    *
   ***
  *****
 *******
*********
```

You could follow the same process you did before and find new expressions that produce the appropriate number of spaces and asterisks. However, there is an easier way. This figure is the same as the previous one, except the lines appear in reverse order. This is a good place to use a decrementing loop to run the `for` loop backward: Instead of starting at `1` and going up to `5` with a `++` update, you can start at `5` and go down to `1` using a `--` update.

The simple way to produce the upward-pointing triangle, then, is with the following code:

```
1  public class DrawCone {
2      public static void main(String[] args) {
3          for (int line = 5; line >= 1; line--) {
4              for (int i = 1; i <= (line - 1); i++) {
5                  System.out.print(" ");
6              }
7              for (int i = 1; i <= (11 - 2 * line); i++) {
8                  System.out.print("*");
9              }
10             System.out.println();
11         }
12     }
13 }
```

Class Constants

The `DrawCone` program in the last section draws a cone with five lines. How would you modify it to produce a cone with three lines? Your first thought might be to simply change the `5` in the code to a `3`. However, that would cause the program to produce the following output:

```
  *****
 *******
*********
```

which is obviously wrong. If you work through the geometry of the figure, you will discover that the problem is with the use of the number `11` in the expression that calculates the number of asterisks to print. The number `11` comes from this formula:

```
2 * (number of lines) + 1
```

Thus, when the number of lines is five, the appropriate value is `11`, but when the number of lines is three, the appropriate value is `7`. Programmers call numbers like these *magic numbers*. They are magic in the sense that they seem to make the program work, but their definition is not always obvious. Glancing at the `DrawCone` program, one is apt to ask, "Why 5? Why 11? Why 3? Why 7? Why me?"

To make programs more readable and more adaptable, you should try to avoid magic numbers whenever possible. You do so by storing the magic numbers. You can use variables to store these values, but that is misleading, given that you are trying to represent values that don't change. Fortunately, Java offers an alternative: You can declare values that are similar to variables but that are guaranteed to have constant values. Not surprisingly, they are called *constants*. We most often define *class constants*, which can be accessed throughout the entire class.

Class Constant

A named value that cannot be changed. A class constant can be accessed anywhere in the class (i.e., its scope is the entire class).

You can choose a descriptive name for a constant that explains what it represents. You can then use that name instead of referring to the specific value to make your programs more readable and adaptable. For example, in the `DrawCone` program, you might want to introduce a constant called `LINES` that represents the number of lines (recall from Chapter 1 that we use all uppercase letters for constant names). You can use that constant in place of the magic number `5` and as part of an expression to calculate a value. This approach allows you to replace the magic number `11` with the formula from which it is derived (`2 * LINES + 1`).

Constants are declared with the keyword `final`, which indicates the fact that their values cannot be changed once assigned, as in

```
final int LINES = 5;
```

You can declare a constant anywhere you can declare a variable, but because constants are often used by several different methods, we generally declare them outside methods. This causes another run-in with our old pal, the `static` keyword. If you want your static methods to be able to access your constants, the constants themselves must be static. Likewise, just as we declare our methods to be public, we usually declare our constants to be public. The following is the general syntax for constant definitions:

```
public static final <type> <name> = <expression>;
```

For example, here are definitions for two constants:

```
public static final int HEIGHT = 10;
public static final int WIDTH = 20;
```

These definitions create constants called `HEIGHT` and `WIDTH` that will always have the values `10` and `20`, respectively. These are known as class constants, because we declare them in the outermost scope of the class, along with the methods of the class. That way, they are visible in each method of the class.

We've already mentioned that we can avoid using a magic number in the `DrawCone` program by introducing a constant for the number of lines. Here's what the constant definition looks like:

```
public static final int LINES = 5;
```

We can now replace the `5` in the outer loop with this constant and replace the `11` in the second inner loop with the expression `2 * LINES + 1`. The result is the following program:

```
1  public class DrawCone2 {
2      public static final int LINES = 5;
3
4      public static void main(String[] args) {
5          for (int line = LINES; line >= 1; line--) {
6              for (int i = 1; i <= (line - 1); i++) {
7                  System.out.print(" ");
8              }
9              int stars = 2 * LINES + 1 - 2 * line;
10             for (int i = 1; i <= stars; i++) {
11                 System.out.print("*");
12             }
13             System.out.println();
14         }
15     }
16 }
```

Notice that in this program the expression for the number of stars has become sufficiently complex that we've introduced a local variable called `stars` to store the value. The advantage of this program is that it is more readable and more adaptable. A simple change to the constant `LINES` will make it produce a figure with a different number of lines.

2.5 Case Study: Hourglass Figure


VideoNote

Now we'll consider an example that is even more complex. To solve it, we will follow three basic steps:

1. Decompose the task into subtasks, each of which will become a static method.
2. For each subtask, make a table for the figure and compute formulas for each column of the table in terms of the line number.
3. Convert the tables into actual `for` loop code for each method.

The output we want to produce is the following:

```
+------+
|\..../|
| \../ |
|  \/  |
|  /\  |
| /..\ |
|/....\|
+------+
```

Problem Decomposition and Pseudocode

To generate this figure, you have to first break it down into subfigures. In doing so, you should look for lines that are similar in one way or another. The first and last lines are exactly the same. The three lines after the first line all fit one pattern, and the three lines after that fit another:

```
+------+
```
line

```
|\..../|
| \../ |
|  \/  |
```
top half

```
|  /\  |
| /..\ |
|/....\|
```
bottom half

```
+------+
```
line

Thus, you can break down the overall problem as follows:

```
draw a solid line.
draw the top half of the hourglass.
draw the bottom half of the hourglass.
draw a solid line.
```

You should solve each subproblem independently. Eventually you'll want to incorporate a class constant to make the program more flexible, but let's first solve the problem without worrying about the use of a constant.

The solid line task can be further specified as

```
write a plus on the output line.
write 6 dashes on the output line.
write a plus on the output line.
go to a new output line.
```

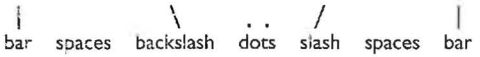
This set of instructions translates easily into a static method:

```
public static void drawLine() {
    System.out.print("+");
    for (int i = 1; i <= 6; i++) {
        System.out.print("-");
    }
    System.out.println("+");
}
```

The top half of the hourglass is more complex. Here is a typical line:

```
| \../ |
```

There are four individual characters, separated by spaces and dots.



Thus, a first approximation in pseudocode might look like this:

```
for (each of 3 lines) {
    write a bar on the output line.
    write some spaces on the output line.
    write a backslash on the output line.
    write some dots on the output line.
    write a slash on the output line.
    write some spaces on the output line.
    write a bar on the output line.
    go to a new line of output.
}
```

Again, you can make a table to figure out the required expressions. Writing the individual characters will be easy enough to translate into Java, but you need to be more specific about the spaces and dots. Each line in this group contains two sets of spaces and one set of dots. Table 2.9 shows how many to use.

The two sets of spaces fit the rule `(line - 1)`, and the number of dots is `(6 - 2 * line)`. Therefore, the pseudocode should read

```
for (line going 1 to 3) {
    write a bar on the output line.
    write (line - 1) spaces on the output line.
    write a backslash on the output line.
    write (6 - 2 * line) dots on the output line.
    write a slash on the output line.
    write (line - 1) spaces on the output line.
    write a bar on the output line.
    go to a new line of output.
}
```

Table 2.9 Analysis of Figure

Line	Spaces	Dots	Spaces
1	0	4	0
2	1	2	1
3	2	0	2

Initial Structured Version

The pseudocode for the top half of the hourglass is easily translated into a static method called `drawTop`. A similar solution exists for the bottom half of the hourglass. Put together, the program looks like this:

```
1  public class DrawFigure {
2      public static void main(String[] args) {
3          drawLine();
4          drawTop();
5          drawBottom();
6          drawLine();
7      }
8
9      // produces a solid line
10     public static void drawLine() {
11         System.out.print("+");
12         for (int i = 1; i <= 6; i++) {
13             System.out.print("-");
14         }
15         System.out.println("+");
16     }
17
18     // produces the top half of the hourglass figure
19     public static void drawTop() {
20         for (int line = 1; line <= 3; line++) {
21             System.out.print("|");
22             for (int i = 1; i <= (line - 1); i++) {
23                 System.out.print(" ");
24             }
25             System.out.print("\\");
26             for (int i = 1; i <= (6 - 2 * line); i++) {
27                 System.out.print(".");
28             }
29             System.out.print("/");
30             for (int i = 1; i <= (line - 1); i++) {
31                 System.out.print(" ");
32             }
```

```
33              System.out.println("|");
34          }
35      }
36
37      // produces the bottom half of the hourglass figure
38      public static void drawBottom() {
39          for (int line = 1; line <= 3; line++) {
40              System.out.print("|");
41              for (int i = 1; i <= (3 - line); i++) {
42                  System.out.print(" ");
43              }
44              System.out.print("/");
45              for (int i = 1; i <= 2 * (line - 1); i++) {
46                  System.out.print(".");
47              }
48              System.out.print("\\");
49              for (int i = 1; i <= (3 - line); i++) {
50                  System.out.print(" ");
51              }
52              System.out.println("|");
53          }
54      }
55  }
```

Adding a Class Constant

The `DrawFigure` program produces the desired output, but it is not very flexible. What if we wanted to produce a similar figure of a different size? The original problem involved an hourglass figure that had three lines in the top half and three lines in the bottom half. What if we wanted the following output, with four lines in the top half and four lines in the bottom half?

```
+--------+
|\....../|
| \..../ |
|  \../  |
|   \/   |
|   /\   |
|  /..\  |
| /....\ |
|/......\|
+--------+
```

Obviously the program would be more useful if we could make it flexible enough to produce either output. We do so by eliminating the magic numbers with the introduction of a class constant. You might think that we need to introduce two constants—one for the height and one for the width—but because of the regularity of this figure,

Table 2.10 Analysis of Different Height Figures

Subheight	Dashes in `drawLine`	Spaces in `drawTop`	Dots in `drawTop`	Spaces in `drawBottom`	Dots in `drawBottom`
3	6	`line - 1`	`6 - 2 * line`	`3 - line`	`2 * (line - 1)`
4	8	`line - 1`	`8 - 2 * line`	`4 - line`	`2 * (line - 1)`

the height is determined by the width and vice versa. Consequently, we only need to introduce a single class constant. Let's use the height of the hourglass halves:

```
public static final int SUB_HEIGHT = 4;
```

We've called the constant `SUB_HEIGHT` rather than `HEIGHT` because it refers to the height of each of the two halves, rather than the figure as a whole. Notice how we use the underscore character to separate the different words in the name of the constant.

So, how do we modify the original program to incorporate this constant? We look through the program for any magic numbers and insert the constant or an expression involving the constant where appropriate. For example, both the `drawTop` and `drawBottom` methods have a `for` loop that executes 3 times to produce 3 lines of output. We change this to 4 to produce 4 lines of output, and more generally, we change it to `SUB_HEIGHT` to produce `SUB_HEIGHT` lines of output.

In other parts of the program we have to update our formulas for the number of dashes, spaces, and dots. Sometimes we can use educated guesses to figure out how to adjust such a formula to use the constant. If you can't guess a proper formula, you can use the table technique to find the appropriate formula. Using this new output with a subheight of 4, you can update the various formulas in the program. Table 2.10 shows the various formulas.

We then go through each formula and figure out how to replace it with a new formula involving the constant. The number of dashes increases by 2 when the subheight increases by 1, so we need a multiplier of 2. The expression `2 * SUB_HEIGHT` produces the correct values. The number of spaces in `drawTop` does not change with the subheight, so the expression does not need to be altered. The number of dots in `drawTop` involves the number 6 for a subheight of 3 and the number 8 for a subheight of 4. Once again we need a multiplier of 2, so we use the expression `2 * SUB_HEIGHT - 2 * line`. The number of spaces in `drawBottom` involves the value 3 for a subheight of 3 and the value 4 for a subheight of 4, so the generalized expression is `SUB_HEIGHT - line`. The number of dots in `drawBottom` does not change when subheight changes.

Here is the new version of the program with a class constant for the subheight. It uses a `SUB_HEIGHT` value of `4`, but we could change this to `3` to produce the smaller version or to some other value to produce yet another version of the figure.

```
1  public class DrawFigure2 {
2      public static final int SUB_HEIGHT = 4;
3
4      public static void main(String[] args) {
5          drawLine();
```

```
6           drawTop();
7           drawBottom();
8           drawLine();
9       }
10
11      // produces a solid line
12      public static void drawLine() {
13          System.out.print("+");
14          for (int i = 1; i <= (2 * SUB_HEIGHT); i++) {
15              System.out.print("-");
16          }
17          System.out.println("+");
18      }
19
20      // produces the top half of the hourglass figure
21      public static void drawTop() {
22          for (int line = 1; line <= SUB_HEIGHT; line++) {
23              System.out.print("|");
24              for (int i = 1; i <= (line - 1); i++) {
25                  System.out.print(" ");
26              }
27              System.out.print("\\");
28              int dots = 2 * SUB_HEIGHT - 2 * line;
29              for (int i = 1; i <= dots; i++) {
30                  System.out.print(".");
31              }
32              System.out.print("/");
33              for (int i = 1; i <= (line - 1); i++) {
34                  System.out.print(" ");
35              }
36              System.out.println("|");
37          }
38      }
39
40      // produces the bottom half of the hourglass figure
41      public static void drawBottom() {
42         for (int line = 1; line <= SUB_HEIGHT; line++) {
43              System.out.print("|");
44              for (int i = 1; i <= (SUB_HEIGHT - line); i++) {
45                  System.out.print(" ");
46              }
47              System.out.print("/");
48              for (int i = 1; i <= 2 * (line - 1); i++) {
49                  System.out.print(".");
50              }
```

```
51              System.out.print("\\");
52              for (int i = 1; i <= (SUB_HEIGHT - line); i++) {
53                  System.out.print(" ");
54              }
55              System.out.println("|");
56          }
57      }
58  }
```

Notice that the `SUB_HEIGHT` constant is declared with class-wide scope, rather than locally in the individual methods. While localizing variables is a good idea, the same is not true for constants. We localize variables to avoid potential interference, but that argument doesn't hold for constants, since they are guaranteed not to change. Another argument for using local variables is that it makes static methods more independent. That argument has some merit when applied to constants, but not enough. It is true that class constants introduce dependencies between methods, but often that is what you want. For example, the three methods in `DrawFigure2` should not be independent of each other when it comes to the size of the figure. Each subfigure has to use the same size constant. Imagine the potential disaster if each method had its own `SUB_HEIGHT`, each with a different value—none of the pieces would fit together.

Further Variations

The solution we have arrived at may seem cumbersome, but it adapts more easily to a new task than does our original program. For example, suppose that you want to generate the following output:

```
+----------+
|\......../|
| \....../ |
|  \..../  |
|   \../   |
|    \/    |
|    /\    |
|   /..\   |
|  /....\  |
| /......\ |
|/........\|
+----------+
|    /\    |
|   /..\   |
|  /....\  |
| /......\ |
|/........\|
|\......../|
| \....../ |
|  \..../  |
|   \../   |
|    \/    |
+----------+
```

This output uses a subheight of 5 and includes both a diamond pattern and an X pattern. You can produce this output by changing the SUB_HEIGHT constant to 5:

```
public static final int SUB_HEIGHT = 5;
```

and rewriting the main method as follows to produce both the original X pattern and the new diamond pattern, which you get simply by reversing the order of the calls on the two halves:

```
public static void main(String[] args) {
    drawLine();
    drawTop();
    drawBottom();
    drawLine();
    drawBottom();
    drawTop();
    drawLine();
}
```

Chapter Summary

Java groups data into types. There are two major categories of data types: primitive data and objects. Primitive types include `int` (integers), `double` (real numbers), `char` (individual text characters), and `boolean` (logical values).

Values and computations are called expressions. The simplest expressions are individual values, also called literals. Some example literals are: `42`, `3.14`, `'Q'`, and `false`. Expressions may contain operators, as in `(3 + 29) - 4 * 5`. The division operation is odd in that it's split into quotient (`/`) and remainder (`%`) operations.

Rules of precedence determine the order in which multiple operators are evaluated in complex expressions. Multiplication and division are performed before addition and subtraction. Parentheses can be used to force a particular order of evaluation.

Data can be converted from one type to another by an operation called a cast.

Variables are memory locations in which values can be stored. A variable is declared with a name and a type. Any data value with a compatible type can be stored in the variable's memory and used later in the program.

Primitive data can be printed on the console using the `System.out.println` method, just like text strings. A string can be connected to another value (concatenated) with the + operator to produce a larger string. This feature allows you to print complex expressions including numbers and text on the console.

A loop is used to execute a group of statements several times. The `for` loop is one kind of loop that can be used to apply the same statements over a range of numbers or to

repeat statements a specified number of times. A loop can contain another loop, called a nested loop.

A variable exists from the line where it is declared to the right curly brace that encloses it. This range, also called the scope of the variable, constitutes the part of the program where the variable can legally be used. A variable declared inside a method or loop is called a local variable. A local variable can only be used inside its method or loop.

An algorithm can be easier to write if you first write an English description of it. Such a description is also called pseudocode.

Important constant values written into a program should be declared as class constants, both to explain their names and values and to make it easier to change their values later.

Self-Check Problems

Section 2.1: Basic Data Concepts

1. Which of the following are legal `int` literals?

```
22   1.5    -1     2.3    10.0   5.     -6875309      '7'
```

2. Trace the evaluation of the following expressions, and give their resulting values:

a. `2 + 3 * 4 - 6`
b. `14 / 7 * 2 + 30 / 5 + 1`
c. `(12 + 3) / 4 * 2`
d. `(238 % 10 + 3) % 7`
e. `(18 - 7) * (43 % 10)`
f. `2 + 19 % 5 - (11 * (5 / 2))`
g. `813 % 100 / 3 + 2.4`
h. `26 % 10 % 4 * 3`
i. `22 + 4 * 2`
j. `23 % 8 % 3`
k. `12 - 2 - 3`
l. `6/2 + 7/3`
m. `6 * 7 % 4`
n. `3 * 4 + 2 * 3`
o. `177 % 100 % 10 / 2`
p. `89 % (5 + 5) % 5`
q. `392 / 10 % 10 / 2`
r. `8 * 2 - 7 / 4`
s. `37 % 20 % 3 * 4`
t. `17 % 10 / 4`

3. Trace the evaluation of the following expressions, and give their resulting values:

a. `4.0 / 2 * 9 / 2`
b. `2.5 * 2 + 8 / 5.0 + 10 / 3`
c. `12 / 7 * 4.4 * 2 / 4`
d. `4 * 3 / 8 + 2.5 * 2`
e. `(5 * 7.0 / 2 - 2.5) / 5 * 2`
f. `41 % 7 * 3 / 5 + 5 / 2 * 2.5`
g. `10.0 / 2 / 4`
h. `8 / 5 + 13 / 2 / 3.0`
i. `(2.5 + 3.5) / 2`
j. `9 / 4 * 2.0 - 5 / 4`
k. `9 / 2.0 + 7 / 3 - 3.0 / 2`
l. `813 % 100 / 3 + 2.4`
m. `27 / 2 / 2.0 * (4.3 + 1.7) - 8 / 3`
n. `53 / 5 / (0.6 + 1.4) / 2 + 13 / 2`
o. `2.5 * 2 + 8 / 5.0 + 10 / 3`
p. `2 * 3 / 4 * 2 / 4.0 + 4.5 - 1`
q. `89 % 10 / 4 * 2.0 / 5 + (1.5 + 1.0 / 2) * 2`

4. Trace the evaluation of the following expressions, and give their resulting values:

a. `2 + 2 + 3 + 4`
b. `"2 + 2" + 3 + 4`
c. `2 + " 2 + 3 " + 4`
d. `3 + 4 + " 2 + 2"`
e. `"2 + 2 " + (3 + 4)`
f. `"(2 + 2) " + (3 + 4)`
g. `"hello 34 " + 2 * 4`
h. `2 + "(int) 2.0" + 2 * 2 + 2`
i. `4 + 1 + 9 + "," + (-3 + 10) + 11 / 3`
j. `8 + 6 * -2 + 4 + "0" + (2 + 5)`
k. `1 + 1 + "8 - 2" + (8 - 2) + 1 + 1`
l. `5 + 2 + "(1 + 1)" + 4 + 2 * 3`
m. `"1" + 2 + 3 + "4" + 5 * 6 + "7" + (8 + 9)`

Section 2.2: Variables

5. Which of the following choices is the correct syntax for declaring a real number variable named `grade` and initializing its value to `4.0`?

a. `int grade : 4.0;`
b. `grade = double 4.0;`
c. `double grade = 4.0;`
d. `grade = 4;`
e. `4.0 = grade;`

6. Imagine you are writing a personal fitness program that stores the user's age, gender, height (in feet or meters), and weight (to the nearest pound or kilogram). Declare variables with the appropriate names and types to hold this information.

7. Imagine you are writing a program that stores a student's year (Freshman, Sophomore, Junior, or Senior), the number of courses the student is taking, and his or her GPA on a 4.0 scale. Declare variables with the appropriate names and types to hold this information.

8. Suppose you have an `int` variable called `number`. What Java expression produces the last digit of the number (the 1s place)?

9. The following program contains 9 mistakes! What are they?

```
1  public class Oops2 {
2      public static void main(String[] args) {
3          int x;
4          System.out.println("x is" x);
5
6          int x = 15.2;    // set x to 15.2
7          System.out.println("x is now + x");
8
9          int y;           // set y to 1 more than x
10         y = int x + 1;
11         System.out.println("x and y are " + x + and + y);
12     }
13 }
```

10. Suppose you have an `int` variable called `number`. What Java expression produces the second-to-last digit of the number (the 10s place)? What expression produces the third-to-last digit of the number (the 100s place)?

11. What are the values of `a`, `b`, and `c` after the following statements?

```
int a = 5;
int b = 10;
int c = b;

a = a + 1;
b = b - 1;
c = c + a;
```

12. What are the values of `first` and `second` at the end of the following code? How would you describe the net effect of the code statements in this exercise?

```
int first = 8;
int second = 19;
first = first + second;
second = first - second;
first = first - second;
```

13. Rewrite the code from the previous exercise to be shorter, by declaring the variables together and by using the special assignment operators (e.g., `+=`, `-=`, `*=`, and `/=`) as appropriate.

14. What are the values of `i`, `j`, and `k` after the following statements?

```
int i = 2;
int j = 3;
int k = 4;
int x = i + j + k;


i = x - i - j;
j = x - j - k;
k = x - i - k;
```

15. What is the output from the following code?

```
int max;
int min = 10;
max = 17 - 4 / 10;
max = max + 6;
min = max - min;
System.out.println(max * 2);
System.out.println(max + min);
System.out.println(max);
System.out.println(min);
```

16. Suppose you have a real number variable x. Write a Java expression that computes the following value y while using the `*` operator only four times:

$$y = 12.3x^4 - 9.1x^3 + 19.3x^2 - 4.6x + 34.2$$

17. The following program redundantly repeats the same expressions many times. Modify the program to remove all redundant expressions using variables of appropriate types.

```
1  public class ComputePay {
2      public static void main(String[] args) {
3          // Calculate pay at work based on hours worked each day
4          System.out.println("My total hours worked:");
5          System.out.println(4 + 5 + 8 + 4);
6
7          System.out.println("My hourly salary:");
8          System.out.println("$8.75");
9
10         System.out.println("My total pay:");
11         System.out.println((4 + 5 + 8 + 4) * 8.75);
```

```
12
13          System.out.println("My taxes owed:");  // 20% tax
14          System.out.println((4 + 5 + 8 + 4) * 8.75 * 0.20);
15      }
16  }
```

Section 2.3: The `for` Loop

18. Complete the following code, replacing the "FINISH ME" parts with your own code:

```
public class Count2 {
    public static void main(String[] args) {
        for (int i = /* FINISH ME */) {
            System.out.println(/* FINISH ME */);
        }
    }
}
```

to produce the following output:

```
2 times 1 = 2
2 times 2 = 4
2 times 3 = 6
2 times 4 = 8
```

19. Assume that you have a variable called `count` that will take on the values `1`, `2`, `3`, `4`, and so on. You are going to formulate expressions in terms of `count` that will yield different sequences. For example, to get the sequence `2`, `4`, `6`, `8`, `10`, `12`, `...`, you would use the expression `(2 * count)`. Fill in the following table, indicating an expression that will generate each sequence.

Sequence	Expression
a. 2, 4, 6, 8, 10, 12, . . .	
b. 4, 19, 34, 49, 64, 79, . . .	
c. 30, 20, 10, 0, −10, −20, . . .	
d. −7, −3, 1, 5, 9, 13, . . .	
e. 97, 94, 91, 88, 85, 82, . . .	

20. Complete the code for the following `for` loop:

```
for (int i = 1; i <= 6; i++) {
    // your code here
}
```

so that it prints the following numbers, one per line:

```
-4
14
32
50
68
86
```

21. What is the output of the following `oddStuff` method?

```
public static void oddStuff() {
    int number = 4;
    for (int count = 1; count <= number; count++) {
        System.out.println(number);
        number = number / 2;
    }
}
```

22. What is the output of the following loop?

```
int total = 25;
for (int number = 1; number <= (total / 2); number++) {
    total = total - number;
    System.out.println(total + " " + number);
}
```

23. What is the output of the following loop?

```
System.out.println("+---+");
for (int i = 1; i <= 3; i++) {
    System.out.println("\\   /");
    System.out.println("/   \\");
}
System.out.println("+---+");
```

24. What is the output of the following loop?

```
for (int i = 1; i <= 3; i++)
    System.out.println("How many lines");
    System.out.println("are printed?");
```

25. What is the output of the following loop?

```
System.out.print("T-minus ");
for (int i = 5; i >= 1; i--) {
    System.out.print(i + ", ");
}
System.out.println("Blastoff!");
```

26. What is the output of the following sequence of loops?

```
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= 10; j++) {
        System.out.print((i * j) + " ");
    }
    System.out.println();
}
```

27. What is the output of the following sequence of loops?

```
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10 - i; j++) {
        System.out.print(" ");
    }
    for (int j = 1; j <= 2 * i - 1; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

28. What is the output of the following sequence of loops?

```
for (int i = 1; i <= 2; i++) {
    for (int j = 1; j <= 3; j++) {
        for (int k = 1; k <= 4; k++) {
            System.out.print("*");
        }
        System.out.print("!");
    }
    System.out.println();
}
```

29. What is the output of the following sequence of loops? Notice that the code is the same as that in the previous exercise, except that the placement of the braces has changed.

```
for (int i = 1; i <= 2; i++) {
    for (int j = 1; j <= 3; j++) {
        for (int k = 1; k <= 4; k++) {
            System.out.print("*");
        }
    }
    System.out.print("!");
    System.out.println();
}
```

30. What is the output of the following sequence of loops? Notice that the code is the same as that in the previous exercise, except that the placement of the braces has changed.

```
for (int i = 1; i <= 2; i++) {
    for (int j = 1; j <= 3; j++) {
```

```
        for (int k = 1; k <= 4; k++) {
            System.out.print("*");
            System.out.print("!");
        }
        System.out.println();
    }
}
```

Section 2.4: Managing Complexity

31. Suppose that you are trying to write a program that produces the following output:

```
1 3 5 7 9 11 13 15 17 19 21
1 3 5 7 9 11
```

The following program is an attempt at a solution, but it contains four major errors. Identify them all.

```
 1  public class BadNews {
 2      public static final int MAX_ODD = 21;
 3
 4      public static void writeOdds() {
 5          // print each odd number
 6          for (int count = 1; count <= (MAX_ODD - 2); count++) {
 7              System.out.print(count + " ");
 8              count = count + 2;
 9          }
10
11          // print the last odd number
12          System.out.print(count + 2);
13      }
14
15      public static void main(String[] args) {
16          // write all odds up to 21
17          writeOdds();
18
19          // now, write all odds up to 11
20          MAX_ODD = 11;
21          writeOdds();
22      }
23  }
```

32. What is the output of the following unknown method?

```
1  public class Strange {
2      public static final int MAX = 5;
3
4      public static void unknown() {
5          int number = 0;
6
```

```
7           for (int count = MAX; count >= 1; count--) {
8               number += (count * count);
9           }
10
11          System.out.println("The result is: " + number);
12      }
13
14      public static void main(String[] args) {
15          unknown();
16      }
17  }
```

33. Suppose that you have a variable called `line` that will take on the values `1, 2, 3, 4,` and so on, and a class constant named `SIZE` that takes one of two values. You are going to formulate expressions in terms of `line` and `SIZE` that will yield different sequences of numbers of characters. Fill in the table below, indicating an expression that will generate each sequence.

`line` value	constant `SIZE` value	Number of characters	Expression
a. 1, 2, 3, 4, 5, 6, ...	1	4, 6, 8, 10, 12, 14, ...	
1, 2, 3, 4, 5, 6, ...	2	6, 8, 10, 12, 14, 16, ...	
b. 1, 2, 3, 4, 5, 6, ...	3	13, 17, 21, 25, 29, 33, ...	
1, 2, 3, 4, 5, 6, ...	5	19, 23, 27, 31, 35, 39, ...	
c. 1, 2, 3, 4, 5, 6, ...	4	10, 9, 8, 7, 6, 5, ...	
1, 2, 3, 4, 5, 6, ...	9	20, 19, 18, 17, 16, 15, ...	

34. Write a table that determines the expressions for the number of each type of character on each of the 6 lines in the following output.

```
!!!!!!!!!!!!!!!!!!!!
\\!!!!!!!!!!!!!!!!!!//
\\\\!!!!!!!!!!!!!!////
\\\\\\!!!!!!!!!!!!//////
\\\\\\\\!!!!!!!!!!////////
\\\\\\\\\\!!!!!!!!!!//////////
```

35. Suppose that a program has been written that produces the output shown in the previous problem. Now the author wants the program to be scalable using a class constant called `SIZE`. The previous output used a constant height of 6, since there were 6 lines. The following is the output for a constant height of 4. Create a new table that shows the expressions for the character counts at this new size of 4, and compare these tables to figure out the expressions for any size using the `SIZE` constant.

```
!!!!!!!!!!!!
\\!!!!!!!!!!//
\\\\!!!!!!!!////
\\\\\\!!!!!!//////
```

Exercises

1. In physics, a common useful equation for finding the position s of a body in linear motion at a given time t, based on its initial position s_0, initial velocity v_0, and rate of acceleration a, is the following:

$$s = s_0 + v_0 t + \frac{1}{2} a t^2$$

Write code to declare variables for s_0, v_0, a, and t, and then write the code to compute s on the basis of these values.

2. Write a `for` loop that produces the following output:

```
1 4 9 16 25 36 49 64 81 100
```

For added challenge, try to modify your code so that it does not need to use the * multiplication operator. (It can be done! Hint: Look at the differences between adjacent numbers.)

3. The Fibonacci numbers are a sequence of integers in which the first two elements are 1, and each following element is the sum of the two preceding elements. The mathematical definition of each *k*th Fibonacci number is the following:

$$F(k) = \begin{cases} F(k-1) + F(k-2), k > 2 \\ 1, k \leq 2 \end{cases}$$

The first 12 Fibonacci numbers are

```
1 1 2 3 5 8 13 21 34 55 89 144
```

Write a `for` loop that computes and prints the first 12 Fibonacci numbers.

4. Write nested `for` loops to produce the following output:

```
*****
*****
*****
*****
```

5. Write nested `for` loops to produce the following output:

```
*
**
***
****
*****
```

6. Write nested `for` loops to produce the following output:

```
1
22
333
4444
55555
666666
7777777
```

7. Write nested `for` loops to produce the following output:

```
    1
   2
  3
 4
5
```

8. Write nested `for` loops to produce the following output:

```
    1
   22
  333
 4444
55555
```

9. Write nested `for` loops to produce the following output, with each line 40 characters wide:

```
----------------------------------------
_-^-_-^-_-^-_-^-_-^-_-^-_-^-_-^-_-^-_-^-
1122334455667788990011223344556677889900
----------------------------------------
```

10. It's common to print a rotating, increasing list of single-digit numbers at the start of a program's output as a visual guide to number the columns of the output to follow. With this in mind, write nested `for` loops to produce the following output, with each line 60 characters wide:

```
        |         |         |         |         |         |
12345678901234567890123456789012345678901234567890123456789012345678901234567890
```

11. Modify your code from the previous exercise so that it could easily be modified to display a different range of numbers (instead of `1234567890`) and a different number of repetitions of those numbers (instead of 60 total characters), with the vertical bars still matching up correctly. Use class constants instead of "magic numbers." Here are some example outputs that could be generated by changing your constants:

```
   |   |   |   |   |   |   |   |   |   |
12340123401234012340123401234012340123401234012340
      |       |       |       |       |       |       |
12345670123456701234567012345670123456701234567012345670
```

12. Write nested `for` loops that produce the following output:

```
000111222333444555666777888999
000111222333444555666777888999
000111222333444555666777888999
```

13. Modify the code so that it now produces the following output:

```
99999888887777766666555554444433333222221111100000
99999888887777766666555554444433333222221111100000
99999888887777766666555554444433333222221111100000
99999888887777766666555554444433333222221111100000
99999888887777766666555554444433333222221111100000
```

14. Modify the code so that it now produces the following output:

```
99999999988888888777777766666655554444333221
99999999988888888777777766666655554444333221
99999999988888888777777766666655554444333221
99999999988888888777777766666655554444333221
```

15. Write a method called `printDesign` that produces the following output. Use nested `for` loops to capture the structure of the figure.

```
-----1-----
----333----
---55555---
--7777777--
-999999999-
```

16. Write a Java program called `SlashFigure` that produces the following output. Use nested `for` loops to capture the structure of the figure. (See also Self-Check Problems 34 and 35.)

```
!!!!!!!!!!!!!!!!!!!!
\\!!!!!!!!!!!!!!!!!!//
\\\\!!!!!!!!!!!!!!!!////
\\\\\\!!!!!!!!!!!!!!//////
\\\\\\\\!!!!!!!!!!!!////////
\\\\\\\\\\!!!!!!!!!!//////////
```

17. Modify your `SlashFigure` program from the previous exercise to become a new program called `SlashFigure2` that uses a global constant for the figure's height. (You may want to make loop tables first.) The previous output used a constant height of 6. The following are the outputs for constant heights of 4 and 8:

Height 4

```
!!!!!!!!!!!!!!
\\!!!!!!!!!!!!//
\\\\!!!!!!!!!!////
\\\\\\!!!!!!!!//////
```

Height 8

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
\\!!!!!!!!!!!!!!!!!!!!!!!!!!!!//
\\\\!!!!!!!!!!!!!!!!!!!!!!!!////
\\\\\\!!!!!!!!!!!!!!!!!!!!//////
\\\\\\\\!!!!!!!!!!!!!!!!////////
\\\\\\\\\\!!!!!!!!!!!!//////////
\\\\\\\\\\\\!!!!!!!!//////////////
\\\\\\\\\\\\\\!!!!!!////////////////
```

18. Write a pseudocode algorithm that will produce the following figure as output:

```
+===+===+
|   |   |
|   |   |
|   |   |
+===+===+
|   |   |
|   |   |
|   |   |
+===+===+
```

19. Use your pseudocode from the previous exercise to write a Java program called `Window` that produces the preceding figure as output. Use nested `for` loops to print the repeated parts of the figure. Once you get it to work, add a class constant so that the size of the figure can be changed simply by changing the constant's value.

20. Write a Java program called `StarFigure` that produces the following output. Use nested for loops to capture the structure of the figure.

```
//////////////\\\\\\\\\\\\\\
///////////*******\\\\\\\\\\\
/////////***************\\\\\\\
////************************\\\\
*******************************
```

21. Modify your `StarFigure` program from the previous exercise to become a new program named `StarFigure2` that uses a global constant for the figure's height. (You may want to make loop tables first.) The previous output used a constant height of 5. The following are the outputs for constant heights of 3 and 6:

Height 3

```
////////\\\\\\\\
////********\\\\
***************
```

Height 6

```
//////////////////////\\\\\\\\\\\\\\\\\\\\\\
////////////////********\\\\\\\\\\\\\\\\\\
//////////***************\\\\\\\\\\\\
////////*************************\\\\\\\\
////*********************************\\\\
***************************************
```

22. Write a Java program called `DollarFigure` that produces the following output. Use nested `for` loops to capture the structure of the figure.

```
$$$$$$$*************$$$$$$$
**$$$$$$***********$$$$$$**
****$$$$$*********$$$$$****
******$$$$*******$$$$******
********$$$*****$$$********
**********$$***$$**********
************$***$************
```

23. Modify your `DollarFigure` program from the previous exercise to become a new program called `DollarFigure2` that uses a global constant for the figure's height. (You may want to make loop tables first.) The previous output used a constant height of 7.

Programming Projects

1. Write a program that produces the following output using nested `for` loops:

```
****** ////////////  ******
*****   ///////////\\   *****
****    ////////\\\\\    ****
***     //////\\\\\\     ***
**      ////\\\\\\\\      **
*       //\\\\\\\\\\       *
        \\\\\\\\\\\\
```

2. Write a program that produces the following output using nested `for` loops:

```
+-------+
|  ^^   |
| ^  ^  |
|^    ^|
|  ^^   |
| ^  ^  |
|^    ^|
+-------+
|v    v|
| v  v  |
|  vv   |
|v    v|
| v  v  |
|  vv   |
+-------+
```

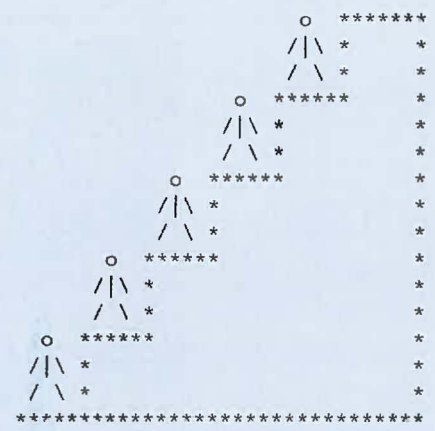
3. Write a program that produces the following output using nested `for` loops:

```
+---------+
|    *    |
|   /*\   |
|  //*\\  |
| ///*\\\ |
| \\\*/// |
|  \\*//  |
|   \*/   |
|    *    |
+---------+
| \\\*/// |
|  \\*//  |
|   \*/   |
|    *    |
|    *    |
|   /*\   |
|  //*\\  |
| ///*\\\ |
+---------+
```

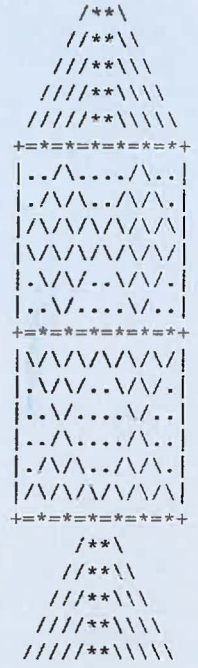
4. Write a program that produces the following hourglass figure as its output using nested `for` loops:

```
|"""""""""|
 \:::::::/
  \:::::/
   \:::/
    |||
   /:::\
  /:::::\
 /:::::::\
|"""""""""|
```

5. Write a program that produces the following output using nested `for` loops. Use a class constant to make it possible to change the number of stairs in the figure.



6. Write a program that produces the following rocket ship figure as its output using nested `for` loops. Use a class constant to make it possible to change the size of the rocket (the following output uses a size of 3).



7. Write a program that produces the following figure (which vaguely resembles the Seattle Space Needle) as its output using nested `for` loops. Use a class constant to make it possible to change the size of the figure (the following output uses a size of 4).