Chapter 4

Conditional Execution

Introduction

In the last few chapters, you've seen how to solve complex programming problems using `for` loops to repeat certain tasks many times. You've also seen how to introduce some flexibility into your programs by using class constants and how to read values input by the user with a `Scanner` object. Now we are going to explore a much more powerful technique for writing code that can adapt to different situations.

In this chapter, we'll look at *conditional execution* in the form of a control structure known as the `if/else` statement. With `if/else` statements, you can instruct the computer to execute different lines of code depending on whether certain conditions are true. The `if/else` statement, like the `for` loop, is so powerful that you will wonder how you managed to write programs without it.

This chapter will also expand your understanding of common programming situations. It includes an exploration of loop techniques that we haven't yet examined and includes a discussion of text-processing issues. Adding conditional execution to your repertoire will also require us to revisit methods, parameters, and return values so that you can better understand some of the fine points. The chapter concludes with several rules of thumb that help us to design better procedural programs.

4.1 if/else Statements

You will often find yourself writing code that you want to execute some of the time but not all of the time. For example, if you are writing a game-playing program, you might want to print a message each time the user achieves a new high score and store that score. You can accomplish this by putting the required two lines of code inside an `if` statement:

```
if (currentScore > maxScore) {
    System.out.println("A new high score!");
    maxScore = currentScore;
}
```

The idea is that you will sometimes want to execute the two lines of code inside the `if` statement, but not always. The test in parentheses determines whether or not the statements inside the `if` statement are executed. In other words, the test describes the conditions under which we want to execute the code.

The general form of the `if` statement is as follows:

```
if (<test>) {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

The `if` statement, like the `for` loop, is a control structure. Notice that we once again see a Java keyword (`if`) followed by parentheses and a set of curly braces enclosing a series of controlled statements.

The diagram in Figure 4.1 indicates the flow of control for the simple `if` statement. The computer performs the test, and if it evaluates to `true`, the computer executes the controlled statements. If the test evaluates to `false`, the computer skips the controlled statements.

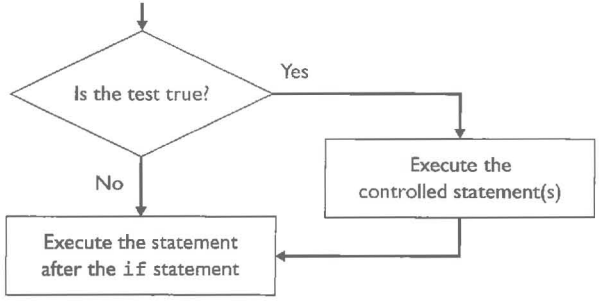


Figure 4.1 Flow of `if` statement

You'll use the simple `if` statement when you have code that you want to execute sometimes and skip other times. Java also has a variation known as the `if/else` statement that allows you to choose between two alternatives. Suppose, for example, that you want to set a variable called `answer` to the square root of a number:

```
answer = Math.sqrt(number);
```

You don't want to ask for the square root if the number is negative. To avoid this potential problem, you could use a simple `if` statement:

```
if (number >= 0) {
    answer = Math.sqrt(number);
}
```

This code will avoid asking for the square root of a negative number, but what value will it assign to `answer` if `number` is negative? In this case, you'll probably want to give a value to `answer` either way. Suppose you want `answer` to be `-1` when `number` is negative. You can express this pair of alternatives with the following `if/else` statement:

```
if (number >= 0) {
    answer = Math.sqrt(number);
} else {
    answer = -1;
}
```

The `if/else` statement provides two alternatives and executes one or the other. So, in the code above, you know that `answer` will be assigned a value regardless of whether `number` is positive or negative.

The general form of the `if/else` statement is:

```
if (<test>) {
    <statement>;
    <statement>;
    ...
    <statement>;
} else {
    <statement>;
    <statement>;
    ...
    <statement>;
}
```

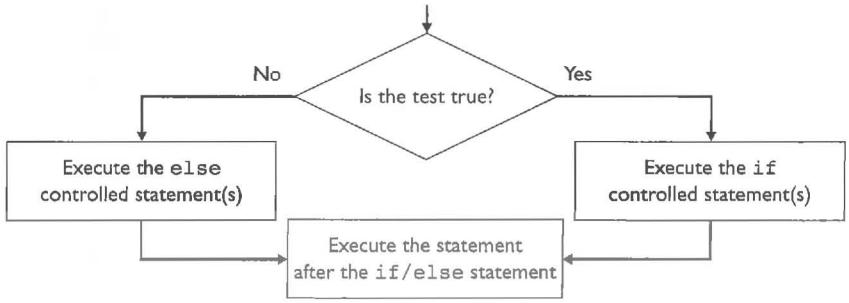


Figure 4.2 Flow of `if/else` statement

This control structure is unusual in that it has two sets of controlled statements and two different keywords (`if` and `else`). Figure 4.2 indicates the flow of control. The computer performs the test and, depending upon whether the code evaluates to `true` or `false`, executes one or the other group of statements.

As in the case of the `for` loop, if you have a single statement to execute, you don't need to include curly braces. However, the Java convention is to include the curly braces even if you don't need them, and we follow that convention in this book.

Relational Operators

An `if/else` statement is controlled by a test. Simple tests compare two expressions to see if they are related in some way. Such tests are themselves expressions of the following form and return either `true` or `false`:

```
<expression> <relational operator> <expression>
```

To evaluate a test of this form, first evaluate the two expressions and then see whether the given relation holds between the value on the left and the value on the right. If the relation holds, the test evaluates to `true`. If not, the test evaluates to `false`.

The relational operators are listed in Table 4.1. Notice that the equality operator consists of two equals signs (==), to distinguish it from the assignment operator (=).

Table 4.1 Relational Operators

Operator	Meaning	Example	Value
==	equal to	`2 + 2 == 4`	`true`
!=	not equal to	`3.2 != 4.1`	`true`
<	less than	`4 < 3`	`false`
>	greater than	`4 > 3`	`true`
<=	less than or equal to	`2 <= 0`	`false`
>=	greater than or equal to	`2.4 >= 1.6`	`true`

Table 4.2 Java Operator Precedence

Description	Operators
unary operators	`++, --, +, -`
multiplicative operators	`*, /, %`
additive operators	`+, -`
relational operators	`<, >, <=, >=`
equality operators	`==, !=`
assignment operators	`=, +=, -=, *=, /=, %=`

Because we use the relational operators as a new way of forming expressions, we must reconsider precedence. Table 4.2 is an updated version of Table 2.5 that includes these new operators. You will see that, technically, the equality comparisons have a slightly different level of precedence than the other relational operators, but both sets of operators have lower precedence than the arithmetic operators.

Let's look at an example. The following expression is made up of the constants `3`, `2`, and `9` and contains addition, multiplication, and equality operations:

```
3 + 2 * 2 == 9
```

Which of the operations is performed first? Because the relational operators have a lower level of precedence than the arithmetic operators, the multiplication is performed first, then the addition, then the equality test. In other words, Java will perform all of the "math" operations first before it tests any relationships. This precedence scheme frees you from the need to place parentheses around the left and right sides of a test that uses a relational operator. When you follow Java's precedence rules, the sample expression is evaluated as follows:

```
3 + 2 * 2 == 9
3 + 4     == 9
    7     == 9
    false
```

You can put arbitrary expressions on either side of the relational operator, as long as the types are compatible. Here is a test with complex expressions on either side:

```
(2 - 3 * 8) / (435 % (7 * 2)) <= 3.8 - 4.5 / (2.2 * 3.8)
```

One limitation of the relational operators is that they should be used only with primitive data. Later in this chapter we will talk about how to compare objects for equality, and in a later chapter we'll discuss how to perform less-than and greater-than comparisons on objects.

Nested if/else Statements


VideoNote

Many beginners write code that looks like this:

```
if (<test1>) {
    <statement1>;
}
if (<test2>) {
    <statement2>;
}
if (<test3>) {
    <statement3>;
}
```

This sequential structure is appropriate if you want to execute any combination of the three statements. For example, you might write this code in a program for a questionnaire with three optional parts, any combination of which might be applicable for a given person.

Figure 4.3 shows the flow of the sequential `if` code. Notice that it's possible for the computer to execute none of the controlled statements (if all tests are false), just one of them (if only one test happens to be true), or more than one of them (if multiple tests are true).

Often, however, you only want to execute one of a series of statements. In such cases, it is better to *nest* the `if` statements, stacking them one inside another:

```
if (<test1>) {
    <statement1>;
} else {
    if (<test2>) {
        <statement2>;
    } else {
        if (<test3>) {
            <statement3>;
        }
    }
}
```

When you use this construct, you can be sure that the computer will execute at most one statement: the statement corresponding to the first test that evaluates to `true`. If no tests evaluate to `true`, no statement is executed. If executing at most

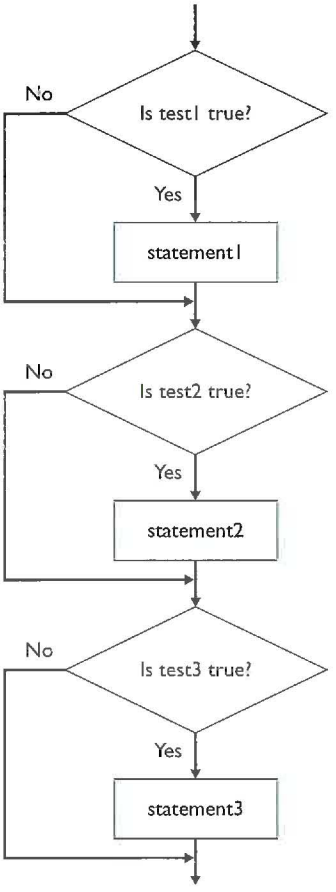


Figure 4.3 Flow of sequential `if`s

one statement is your objective, this construct is more appropriate than the sequential `if` statements. It reduces the likelihood of errors and simplifies the testing process.

As you can see, nesting `if` statements like this leads to a lot of indentation. The indentation isn't very helpful, because this construct is really intended to allow the choice of one of a number of alternatives. K&R style has a solution for this as well. If an `else` is followed by an `if`, we put them on the same line:

```
if (<test1>) {
    <statement1>;
} else if (<test2>) {
    <statement2>;
} else if (<test3>) {
    <statement3>;
}
```

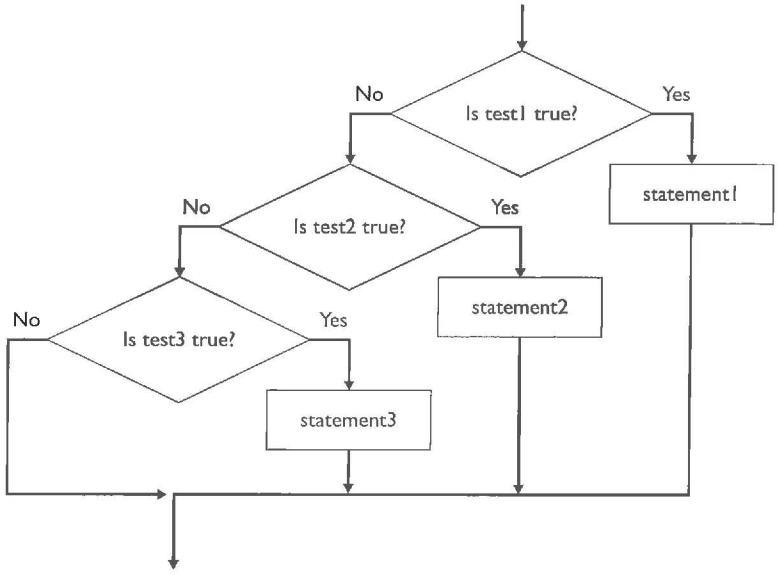


Figure 4.4 Flow of nested `if`s ending in test

When you follow this convention, the various statements all appear at the same level of indentation. We recommend that nested `if/else` statements be indented in this way.

Figure 4.4 shows the flow of the nested `if/else` code. Notice that it is possible to execute one of the controlled statements (the first one that evaluates to `true`) or none (if no tests evaluate to `true`).

In a variation of this structure, the final statement is controlled by an `else` instead of a test:

```
if (<test1>) {
    <statement1>;
} else if (<test2>) {
    <statement2>;
} else {
    <statement3>;
}
```

In this construct, the computer will always select the final branch when all the tests fail, and thus the construct will always execute exactly one of the three statements. Figure 4.5 shows the flow of this modified nested `if/else` code.

To explore these variations, consider the task of having the computer state whether a number is positive, negative, or zero. You could structure this task as three simple `if` statements as follows:

```
if (number > 0) {
    System.out.println("Number is positive.");
}
if (number == 0) {
    System.out.println("Number is zero.");
}
if (number < 0) {
    System.out.println("Number is negative.");
}
```

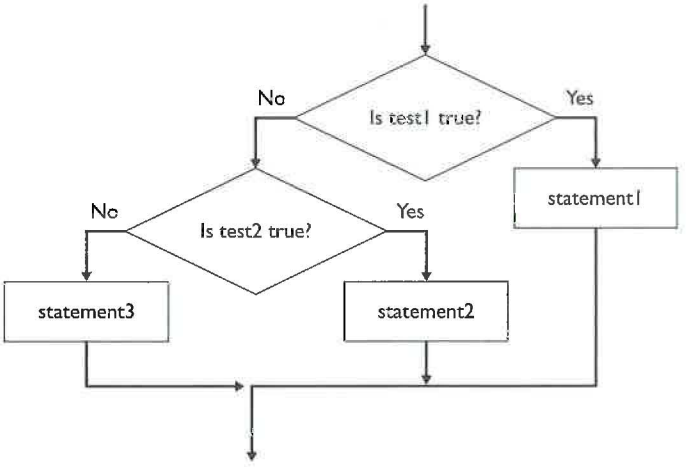


Figure 4.5 Flow of nested `if`s ending in `else`

To determine how many of the `println`s are potentially executed, you have to stop and think about the tests being performed. But you shouldn't have to put that much effort into understanding this code. The code is clearer if you nest the `if` statements:

```
if (number > 0) {
    System.out.println("Number is positive.");
} else if (number == 0) {
    System.out.println("Number is zero.");
} else if (number < 0) {
    System.out.println("Number is negative.");
}
```

This solution has a problem, however. You know that you want to execute one and only one `println` statement, but this nested structure does not preclude the possibility of no statement being executed (which would happen if all three tests failed). Of course, with these particular tests that will never happen: If a number is neither positive nor zero, it must be negative. Thus, the final test here is unnecessary and

misleading. You must think about the tests to determine whether or not it is possible for all three tests to fail and all three branches to be skipped.

In this case, the best solution is the nested `if/else` approach with a final branch that is always taken if the first two tests fail:

```
if (number > 0) {
    System.out.println("Number is positive.");
} else if (number == 0) {
    System.out.println("Number is zero.");
} else {
    System.out.println("Number is negative.");
}
```

You can glance at this construct and see immediately that exactly one `println` will be executed. You don't have to look at the tests being performed in order to realize this; it is a property of this kind of nested `if/else` structure. If you want, you can include a comment to make it clear what is going on:

```
if (number > 0) {
    System.out.println("Number is positive.");
} else if (number == 0) {
    System.out.println("Number is zero.");
} else { // number must be negative
    System.out.println("Number is negative.");
}
```

One final benefit of this approach is efficiency. When the code includes three simple `if` statements, the computer will always perform all three tests. When the code uses the nested `if/else` approach, the computer carries out tests only until a match is found, which is a better use of resources. For example, in the preceding code we only need to perform one test for positive numbers and at most two tests overall.

When you find yourself writing code to choose among alternatives like these, you have to analyze the particular problem to figure out how many of the branches you potentially want to execute. If it doesn't matter what combination of branches is taken, use sequential `if` statements. If you want one or none of the branches to be taken, use nested `if/else` statements with a test for each statement. If you want exactly one branch to be taken, use nested `if/else` statements with a final branch controlled by an `else` rather than by a test. Table 4.3 summarizes these choices.

Table 4.3 `if/else` Options

Situation	Construct	Basic form

You want to execute any combination of controlled statements	Sequential `if`s	`if (<test1>) {`
		`  <statement1>;`
		`}`
		`if (<test2>) {`
		`  <statement2>;`
		`}`
		`if (<test3>) {`
		`  <statement3>;`
		`}`

You want to execute zero or one of the controlled statements	Nested `if`s ending in test	`if (<test1>) {`
		`  <statement1>;`
		`} else if (<test2>) {`
		`  <statement2>;`
		`} else if (<test3>) {`
		`  <statement3>;`
		`}`

You want to execute exactly one of the controlled statements	Nested `if`s ending in `else`	`if (<test1>) {`
		`  <statement1>;`
		`} else if (<test2>) {`
		`  <statement2>;`
		`} else {`
		`  <statement3>;`
		`}`

Common Programming Error

Choosing the Wrong `if/else` Construct

Suppose that your instructor has told you that grades will be determined as follows:

A for scores ≥ 90
B for scores ≥ 80
C for scores ≥ 70
D for scores ≥ 60
F for scores < 60

You can translate this scale into code as follows:

```
String grade;
if (score >= 90) {
    grade = "A";
```

Continued on next page

Continued from previous page

```
}
if (score >= 80) {
    grade = "B";
}
if (score >= 70) {
    grade = "C";
}
if (score >= 60) {
    grade = "D";
}
if (score < 60) {
    grade = "F";
}
```

However, if you then try to use the variable `grade` after this code, you'll get this error from the compiler:

```
variable grade might not have been initialized
```

This is a clue that there is a problem. The Java compiler is saying that it believes there are paths through this code that will leave the variable `grade` uninitialized. In fact, the variable will always be initialized, but the compiler cannot figure this out. We can fix this problem by giving an initial value to `grade`:

```
String grade = "no grade";
```

This change allows the code to compile. But if you compile and run the program, you will find that it gives out only two grades: D and F. Anyone who has a score of at least 60 ends up with a D and anyone with a grade below 60 ends up with an F. And even though the compiler complained that there was a path that would allow `grade` not to be initialized, no one ever gets a grade of "no grade."

The problem here is that you want to execute exactly one of the assignment statements, but when you use sequential `if` statements, it's possible for the program to execute several of them sequentially. For example, if a student has a score of 95, that student's `grade` is set to `"A"`, then reset to `"B"`, then reset to `"C"`, and finally reset to `"D"`. You can fix this problem by using a nested `if/else` construct:

```
String grade;
if (score >= 90) {
    grade = "A";
} else if (score >= 80) {
```

Continued on next page

Continued from previous page

```
    grade = "B";
} else if (score >= 70) {
    grade = "C";
} else if (score >= 60) {
    grade = "D";
} else { // score < 60
    grade = "F";
}
```

You don't need to set `grade` to `"no grade"` now because the compiler can see that no matter what path is followed, the variable `grade` will be assigned a value (exactly one of the branches will be executed).

Object Equality

You saw earlier in the chapter that you can use the == and != operators to test for equality and nonequality of primitive data, respectively. Unfortunately, these operators do not work the way you might expect when you test for equality of objects like strings. You will have to learn a new way to test objects for equality.

For example, you might write code like the following to read a token from the console and to call one of two different methods depending on whether the user responded with "yes" or "no." If the user types neither word, this code is supposed to print an error message:

```
System.out.print("yes or no? ");
String s = console.next();
if (s == "yes") {
    processYes();
} else if (s == "no") {
    processNo();
} else {
    System.out.println("You didn't type yes or no");
}
```

Unfortunately, this code does not work. No matter what the user enters, this program always prints "You didn't type yes or no". We will explore in detail in Chapter 8 why this code doesn't work. For now the important thing to know is that Java provides a second way of testing for equality that is intended for use with objects. Every Java object has a method called `equals` that takes another object as an argument. You can use this method to ask an object whether it equals another object. For example, we can fix the previous code as follows:

```
System.out.print("yes or no? ");
String s = console.next();
```

```
if (s.equals("yes")) {
    processYes();
} else if (s.equals("no")) {
    processNo();
} else {
    System.out.println("You didn't type yes or no");
}
```

Remember when you're working with strings that you should always call the `equals` method rather than using ==.

The `String` class also has a special variation of the `equals` method called `equalsIgnoreCase` that ignores case differences (uppercase versus lowercase letters). For example, you could rewrite the preceding code as follows to recognize responses like "Yes," "YES," "No," "NO," yES", and so on:

```
System.out.print("yes or no? ");
String s = console.next();
if (s.equalsIgnoreCase("yes")) {
    processYes();
} else if (s.equalsIgnoreCase("no")) {
    processNo();
} else {
    System.out.println("You didn't type yes or no");
}
```

Factoring `if/else` Statements

Suppose you are writing a program that plays a betting game with a user and you want to give different warnings about how much cash the user has left. The nested `if/else` construct that follows distinguishes three different cases: funds less than $500, which is considered low; funds between $500 and $1000, which is considered okay; and funds over $1000, which is considered good. Notice that the user is given different advice in each case:

```
if (money < 500) {
    System.out.println("You have $" + money + " left.");
    System.out.print("Cash is dangerously low. Bet carefully.");
    System.out.print("How much do you want to bet? ");
    bet = console.nextInt();
} else if (money < 1000) {
    System.out.println("You have $" + money + " left.");
    System.out.print("Cash is somewhat low. Bet moderately.");
    System.out.print("How much do you want to bet? ");
    bet = console.nextInt();
} else {
```

```
    System.out.println("You have $" + money + " left.");
    System.out.print("Cash is in good shape. Bet liberally.");
    System.out.print("How much do you want to bet? ");
    bet = console.nextInt();
}
```

This construct is repetitious and can be made more efficient by using a technique called *factoring*. Using this simple technique, you factor out common pieces of code from the different branches of the `if/else` construct. In the preceding program, three different branches can execute, depending on the value of the variable `money`. Start by writing down the series of actions being performed in each branch and comparing them, as in Figure 4.6.

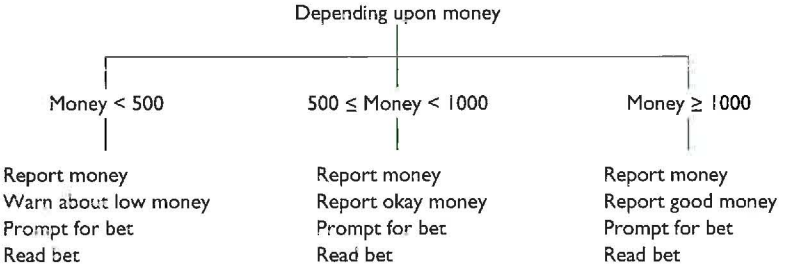


Figure 4.6 `if/else` branches before factoring

You can factor at both the top and the bottom of a construct like this. If you notice that the top statement in each branch is the same, you factor it out of the branching part and put it before the branch. Similarly, if the bottom statement in each branch is the same, you factor it out of the branching part and put it after the loop. You can factor the top statement in each of these branches and the bottom two statements, as in Figure 4.7.

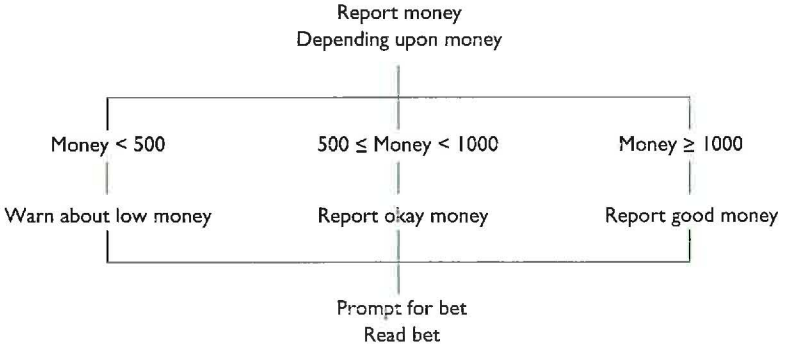


Figure 4.7 `if/else` branches after factoring

Thus, the preceding code can be reduced to the following more succinct version:

```
System.out.println("You have $" + money + " left.");
if (money < 500) {
```

```
    System.out.print("Cash is dangerously low. Bet carefully.");
} else if (money < 1000) {
    System.out.print("Cash is somewhat low. Bet moderately.");
} else {
    System.out.print("Cash is in good shape. Bet liberally.");
}
System.out.print("How much do you want to bet? ");
bet = console.nextInt();
```

Testing Multiple Conditions

When you are writing a program, you often find yourself wanting to test more than one condition. For example, suppose you want the program to take a particular course of action if a number is between 1 and 10. You might say:

```
if (number >= 1) {
    if (number <= 10) {
        doSomething();
    }
}
```

In these lines of code, you had to write two statements: one testing whether the number was greater than or equal to 1 and one testing whether the number was less than or equal to 10.

Java provides an efficient alternative: You can combine the two tests by using an operator known as the logical AND operator, which is written as two ampersands with no space in between (`&&`). Using the AND operator, we can write the preceding code more simply:

```
if (number >= 1 && number <= 10) {
    doSomething();
}
```

As its name implies, the AND operator forms a test that requires that both parts of the test evaluate to `true`. There is a similar operator known as logical OR that evaluates to `true` if either of two tests evaluates to `true`. The logical OR operator is written using two vertical bar characters (`||`). For example, if you want to test whether a variable `number` is equal to 1 or 2, you can say:

```
if (number == 1 || number == 2) {
    processNumber(number);
}
```

We will explore the logical AND and logical OR operators in more detail in the next chapter.

4.2 Cumulative Algorithms

The more you program, the more you will find that certain patterns emerge. Many common algorithms involve accumulating an answer step by step. In this section, we will explore some of the most common *cumulative algorithms*.

> **Cumulative Algorithm**
>
> An operation in which an overall value is computed incrementally, often using a loop.

For example, you might use a cumulative algorithm over a set of numbers to compute the average value or to find the largest number.

Cumulative Sum

You'll often want to find the sum of a series of numbers. One way to do this is to declare a different variable for each value you want to include, but that would not be a practical solution: If you have to add a hundred numbers together, you won't want to declare a hundred different variables. Fortunately, there is a simpler way.

The trick is to keep a running tally of the result and process one number at a time. For example, to add to a variable called `sum`, you would write the following line of code:

```
sum = sum + next;
```

Alternatively, you could use the shorthand assignment operator:

```
sum += next;
```

The preceding statement takes the existing value of `sum`, adds the value of a variable called `next`, and stores the result as the new value of `sum`. This operation is performed for each number to be summed. Notice that when you execute this statement for the first time `sum` does not have a value. To get around this, you initialize `sum` to a value that will not affect the answer: `0`.

Here is a pseudocode description of the cumulative sum algorithm:

```
sum = 0.
for (all numbers to sum) {
    obtain "next".
    sum += next.
}
```

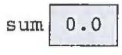
To implement this algorithm, you must decide how many times to go through the loop and how to obtain a `next` value. Here is an interactive program that prompts the user for the number of numbers to add together and for the numbers themselves:

```
1 // Finds the sum of a sequence of numbers.
2
3 import java.util.*;
4
5 public class ExamineNumbers1 {
6     public static void main(String[] args) {
7         System.out.println("This program adds a sequence of");
8         System.out.println("numbers.");
9         System.out.println();
10
11        Scanner console = new Scanner(System.in);
12
13        System.out.print("How many numbers do you have? ");
14        int totalNumber = console.nextInt();
15
16        double sum = 0.0;
17        for (int i = 1; i <= totalNumber; i++) {
18            System.out.print("    #" + i + "? ");
19            double next = console.nextDouble();
20            sum += next;
21        }
22        System.out.println();
23
24        System.out.println("sum = " + sum);
25    }
26 }
```

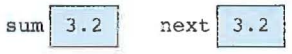
The program's execution will look something like this (as usual, user input is boldface):

```
This program adds a sequence of
numbers.

How many numbers do you have? 6
    #1? 3.2
    #2? 4.7
    #3? 5.1
    #4? 9.0
    #5? 2.4
    #6? 3.1

sum = 27.5
```

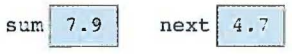
Let's trace the execution in detail. Before we enter the `for` loop, we initialize the variable `sum` to `0.0`:



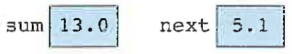
On the first execution of the `for` loop, we read in a value of `3.2` from the user and add this value to `sum`:



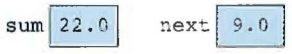
The second time through the loop, we read in a value of `4.7` and add this to the value of `sum`:



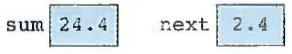
Notice that the `sum` now includes both of the numbers entered by the user, because we have added the new value, `4.7`, to the old value, `3.2`. The third time through the loop, we add in the value `5.1`:



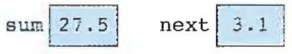
Notice that the variable `sum` now contains the sum of the first three numbers (`3.2 + 4.7 + 5.1`). Now we read in `9.0` and add it to the sum:



Then we add in the fifth value, `2.4`:



Finally, we add in the sixth value, `3.1`:



We then exit the `for` loop and print the value of `sum`.

There is an interesting scope issue in this particular program. Notice that the variable `sum` is declared outside the loop, while the variable `next` is declared inside the loop. We have no choice but to declare `sum` outside the loop because it needs to be initialized and it is used after the loop. But the variable `next` is used only inside the loop, so it can be declared in that inner scope. It is best to declare variables in the innermost scope possible.

The cumulative sum algorithm and variations on it will be useful in many of the programming tasks you solve. How would you do a cumulative product? Here is the pseudocode:

```
product = 1.
for (all numbers to multiply) {
    obtain "next".
    product = product * next.
}
```

Min/Max Loops

Another common programming task is to keep track of the maximum and/or minimum values in a sequence. For example, consider the task of deciding whether it will be viable to build a living area on the Moon inhabited by humans. One obstacle is

that the average daily surface temperature on the Moon is a chilly −50 degrees Fahrenheit. But a much more daunting problem is the wide range of values; it ranges from a minimum of −240 degrees to a maximum of 250 degrees.

To compute the maximum of a sequence of values, you can keep track of the largest value you've seen so far and use an `if` statement to update the maximum if you come across a new value that is larger than the current maximum. This approach can be described in pseudocode as follows:

```
initialize max.
for (all numbers to examine) {
    obtain "next".
    if (next > max) {
        max = next.
    }
}
```

Initializing the maximum isn't quite as simple as it sounds. For example, novices often initialize `max` to `0`. But what if the sequence of numbers you are examining is composed entirely of negative numbers? For example, you might be asked to find the maximum of this sequence:

−84, −7, −14, −39, −410, −17, −41, −9

The maximum value in this sequence is –7, but if you've initialized `max` to `0`, the program will incorrectly report `0` as the maximum.

There are two classic solutions to this problem. First, if you know the range of the numbers you are examining, you can make an appropriate choice for `max`. In that case, you can set `max` to the lowest value in the range. That seems counterintuitive because normally we think of the maximum as being large, but the idea is to set `max` to the smallest possible value it could ever be so that anything larger will cause `max` to be reset to that value. For example, if you knew that the preceding sequence of numbers consisted of temperatures in degrees Fahrenheit, you would know that they temperatures could never be smaller than absolute zero (around −460 degrees Fahrenheit), so you could initialize `max` to that value.

The second possibility is to initialize `max` to the first value in the sequence. That won't always be convenient because it means obtaining one of the numbers outside the loop.

When you combine these two possibilities, the pseudocode becomes:

```
initialize max either to lowest possible value or to first value.
for (all numbers to examine) {
    obtain "next".
    if (next > max) {
        max = next;
    }
}
```

The pseudocode for computing the minimum is a slight variation of this code:

```
initialize min either to highest possible value or to first value.
for (all numbers to examine) {
    obtain "next".
    if (next < min) {
        min = next.
    }
}
```

To help you understand this better, let's put the pseudocode into action with a real problem. In mathematics, there is an open problem that involves what are known as *hailstone sequences*. These sequences of numbers often rise and fall in unpredictable patterns, which is somewhat analogous to the process that forms hailstones.

A hailstone sequence is a sequence of numbers in which each value x is followed by:

$(3x + 1)$, if x is odd

$\left(\frac{x}{2}\right)$, if x is even

For example, if you start with 7 and construct a sequence of length 10, you get the sequence:

7, 22, 11, 34, 17, 52, 26, 13, 40, 20

In this sequence, the maximum and minimum values are 52 and 7, respectively. If you extend this computation to a sequence of length 20, you get the sequence:

7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1

In this case, the maximum and minimum values are 52 and 1, respectively.

You will notice that once 1, 2, or 4 appears in the sequence, the sequence repeats itself. It is conjectured that all integers eventually reach 1, like hailstones that fall to the ground. This is an unsolved problem in mathematics. Nobody has been able to disprove it, but nobody has proven it either.

Let's write a method that takes a starting value and a sequence length and prints the maximum and minimum values obtained in a hailstone sequence formed with that starting value and of that length. Our method will look like this:

```
public static void printHailstoneMaxMin(int value, int length) {
    ...
}
```

We can use the starting value to initialize `max` and `min`:

```
int min = value;
int max = value;
```

We then need a loop that will generate the other values. The user will input a parameter telling us how many times to go through the loop, but we don't want to execute the loop body `length` times: Remember that the starting value is part of the sequence, so if we want to use a sequence of the given length, we have to make sure that the number of iterations is one less than `length`. Combining this idea with the `max/min` pseudocode, we know the loop will look like this:

```
for (int i = 1; i <= length - 1; i++) {
    compute next number.
    if (value > max) {
        max = value.
    } else if (value < min) {
        min = value.
    }
}
print max and min.
```

To fill out the pseudocode for "compute next number," we need to translate the hailstone formula into code. The formula is different, depending on whether the current value is odd or even. We can use an `if/else` statement to solve this task. For the test, we can use a "mod 2" test to see what remainder we get when we divide by 2. Even numbers have a remainder of 0 and odd numbers have a remainder of 1. So the test should look like this:

```
if (value % 2 == 0) {
    do even computation.
} else {
    do odd computation.
}
```

Translating the hailstone mathematical formulas into Java, we get the following code:

```
if (value % 2 == 0) {
    value = value / 2;
} else {
    value = 3 * value + 1;
}
```

The only part of our pseudocode that we haven't filled in yet is the part that prints the result. This part comes after the loop and is fairly easy to complete. Here is the complete method:

```
public static void printHailstoneMaxMin(int value, int length) {
    int min = value;
    int max = value;
```

```
    for (int i = 1; i <= length - 1; i++) {
        if (value % 2 == 0) {
            value = value / 2;
        } else {
            value = 3 * value + 1;
        }
        if (value > max) {
            max = value;
        } else if (value < min) {
            min = value;
        }
    }
    System.out.println("max = " + max);
    System.out.println("min = " + min);
}
```

Cumulative Sum with `if`

Let's now explore how you can use `if/else` statements to create some interesting variations on the cumulative sum algorithm. Suppose you want to read a sequence of numbers and compute the average. This task seems like a straightforward variation of the cumulative sum code. You can compute the average as the sum divided by the number of numbers:

```
double average = sum / totalNumber;
System.out.println("average = " + average);
```

But there is one minor problem with this code. Suppose that when the program asks the user how many numbers to process, the user enters `0`. Then the program will not enter the cumulative sum loop, and your code will try to compute the value of `0` divided by `0`. Java will then print that the average is `NaN`, a cryptic message that is short for "Not a Number." It would be better for the program to print out some other kind of message which indicates that there aren't any numbers to average. You can use an `if/else` statement for this purpose:

```
if (totalNumber <= 0) {
    System.out.println("No numbers to average");
} else {
    double average = sum / totalNumber;
    System.out.println("average = " + average);
}
```

Another use of `if` statements would be to count how many negative numbers the user enters. You will often want to count how many times something occurs in a program.

This goal is easy to accomplish with an `if` statement and an integer variable called a *counter*. You start by initializing the counter to `0`:

```
int negatives = 0;
```

You can use any name you want for the variable. Here we used `negatives` because that is what you're counting. The other essential step is to increment the counter inside the loop if it passes the test of interest:

```
if (next < 0) {
    negatives++;
}
```

When you put this all together and modify the comments and introduction, you end up with the following variation of the cumulative sum program:

```
1 // Finds the average of a sequence of numbers as well as
2 // reporting how many of the user-specified numbers were negative.
3
4 import java.util.*;
5
6 public class ExamineNumbers2 {
7     public static void main(String[] args) {
8         System.out.println("This program examines a sequence");
9         System.out.println("of numbers to find the average");
10        System.out.println("and count how many are negative.");
11        System.out.println();
12
13        Scanner console = new Scanner(System.in);
14
15        System.out.print("How many numbers do you have? ");
16        int totalNumber = console.nextInt();
17
18        int negatives = 0;
19        double sum = 0.0;
20        for (int i = 1; i <= totalNumber; i++) {
21            System.out.print("    #" + i + "? ");
22            double next = console.nextDouble();
23            sum += next;
24            if (next < 0) {
25                negatives++;
26            }
27        }
28        System.out.println();
29
30        if (totalNumber <= 0) {
```

```
31              System.out.println("No numbers to average");
32          } else {
33              double average = sum / totalNumber;
34              System.out.println("average = " + average);
35          }
36          System.out.println("# of negatives = " + negatives);
37      }
38  }
```

The program's execution will look something like this:

```
This program examines a sequence
of numbers to find the average
and count how many are negative.

How many numbers do you have? 8
    #1? 2.5
    #2? 9.2
    #3? -19.4
    #4? 208.2
    #5? 42.3
    #6? 92.7
    #7? -17.4
    #8? 8

average = 40.7625
# of negatives = 2
```

Roundoff Errors

As you explore cumulative algorithms, you'll discover a particular problem that you should understand. For example, consider the following execution of the previous `ExamineNumbers2` program with different user input:

```
This program examines a sequence
of numbers to find the average
and count how many are negative.

How many numbers do you have? 4
    #1? 2.1
    #2? -3.8
    #3? 5.4
    #4? 7.4

average = 2.7750000000000004
# of negatives = 1
```

If you use a calculator, you will find that the four numbers add up to 11.1. If you divide this number by 4, you get 2.775. Yet Java reports the result as `2.7750000000000004`. Where do all of those zeros come from, and why does the number end in 4? The answer is that floating-point numbers can lead to *roundoff errors.*

Roundoff Error

A numerical error that occurs because floating-point numbers are stored as approximations rather than as exact values.

Roundoff errors are generally small and can occur in either direction (slightly high or slightly low). In the previous case, we got a roundoff error that was slightly high.

Floating-point numbers are stored in a format similar to scientific notation, with a set of digits and an exponent. Consider how you would store the value one-third in scientific notation using base-10. You would state that the number is 3.33333 (repeating) times 10 to the –1 power. We can't store an infinite number of digits on a computer, though, so we'll have to stop repeating the 3s at some point. Suppose we can store 10 digits. Then the value for one-third would be stored as 3.333333333 times 10 to the –1. If we multiply that number by 3, we don't get back 1. Instead, we get 9.999999999 times 10 to the –1 (which is equal to 0.9999999999).

You might wonder why the numbers we used in the previous example caused a problem when they didn't have any repeating digits. You have to remember that the computer stores numbers in base-2. Numbers like 2.1 and 5.4 might look like simple numbers in base-10, but they have repeating digits when they are stored in base-2.

Roundoff errors can lead to rather surprising outcomes. For example, consider the following short program:

```
1 public class Roundoff {
2     public static void main(String[] args) {
3         double n = 1.0;
4         for (int i = 1; i <= 10; i++) {
5             n += 0.1;
6             System.out.println(n);
7         }
8     }
9 }
```

This program presents a classic cumulative sum with a loop that adds 0.1 to the number `n` each time the loop executes. We start with `n` equal to `1.0` and the loop iterates 10 times, which we might expect to print the numbers 1.1, 1.2, 1.3, and so on through 2.0. Instead, it produces the following output:

```
1.1
1.2000000000000002
1.3000000000000003
```

```
1.4000000000000004
1.5000000000000004
1.6000000000000005
1.7000000000000006
1.8000000000000007
1.9000000000000008
2.000000000000001
```

The problem occurs because 0.1 cannot be stored exactly in base-2 (it produces a repeating set of digits, just as one-third does in base-10). Each time through the loop the error is compounded, which is why the roundoff error gets worse each time.

As another example, consider the task of adding together the values of a penny, a nickel, a dime, and a quarter. If we use variables of type `int`, we will get an exact answer regardless of the order in which we add the numbers:

```
int cents1 = 1 + 5 + 10 + 25;
int cents2 = 25 + 10 + 5 + 1;
System.out.println(cents1);
System.out.println(cents2);
```

The output of this code is as follows:

```
41
41
```

Regardless of the order, these numbers always add up to 41 cents. But suppose that instead of thinking of these values as whole cents, we think of them as fractions of a dollar that we store as `doubles`:

```
double dollars1 = 0.01 + 0.05 + 0.10 + 0.25;
double dollars2 = 0.25 + 0.10 + 0.05 + 0.01;
System.out.println(dollars1);
System.out.println(dollars2);
```

This code has surprising output:

```
0.41000000000000003
0.41
```

Even though we are adding up exactly the same numbers, the fact that we add them in a different order makes a difference. The reason is roundoff errors.

There are several lessons to draw from this:

- Be aware that when you store floating-point values (e.g., `doubles`), you are storing approximations and not exact values. If you need to store an exact value, store it using type `int`.

- Don't be surprised when you see numbers that are slightly off from the expected values.
- Don't expect to be able to compare variables of type `double` for equality.

To follow up on the third point, consider what the preceding code would lead to if we were to perform the following test:

```
if (dollars1 == dollars2) {
    ...
}
```

The test would evaluate to `false` because the values are very close, but not close enough for Java to consider them equal. We rarely use a test for exact equality when we work with `doubles`. Instead, we can use a test like this to see if numbers are close to one another:

```
if (Math.abs(dollars1 - dollars2) < 0.001) {
    ...
}
```

We use the absolute value (`abs`) method from the `Math` class to find the magnitude of the difference and then test whether it is less than some small amount (in this case, `0.001`).

Later in this chapter we'll introduce a variation on `print/println` called `printf` that will make it easier to print numbers like these without all of the extra digits.

4.3 Text Processing

Programmers commonly face problems that require them to create, edit, examine, and format text. Collectively, we call these tasks *text processing*.

> **Text Processing**
>
> Editing and formatting strings of text.

In this section, we'll look in more detail at the `char` primitive type and introduce a new command called `System.out.printf`. Both of these tools are very useful for text-processing tasks.

The `char` Type

The primitive type `char` represents a single character of text. It's legal to have variables, parameters, and return values of type `char` if you so desire. Literal values of type `char` are expressed by placing the character within single quotes:

```
char ch = 'A';
```

Table 4.4 Differences between `char` and `String`

	`char`	`String`
Type of value	primitive	object
Memory usage	2 bytes	depends on length
Methods	none	`length`, `toUpperCase`, ...
Number of letters	exactly 1	0 to many
Surrounded by	apostrophes: `'c'`	quotes: `"Str"`
Comparing	<, >=, ==, ...	`equals`

It is also legal to create a `char` value that represents an escape sequence:

```
char newline = '\n';
```

In the previous chapter we discussed `String` objects. The distinction between `char` and `String` is a subtle one that confuses many new Java programmers. The main difference is that a `String` is an object, but a `char` is a primitive value. A `char` occupies a very small amount of memory, but it has no methods. Table 4.4 summarizes several of the differences between the types.

Why does Java have two types for such similar data? The `char` type exists primarily for historical reasons; it dates back to older languages such as C that influenced the design of Java.

So why would a person ever use the `char` type when `String` is available? It's often necessary to use `char` because some methods in Java's API use it as a parameter or return type. But there are also a few cases in which using `char` can be more useful or simpler than using `String`.

The characters of a `String` are stored inside the object as values of type `char`. You can access the individual characters through the object's `charAt` method, which accepts an integer index as a parameter and returns the character at that index. We often loop over a string to examine or change its characters. For example, the following method prints each character of a string on its own line:

```
public static void printVertical(String message) {
    for (int i = 0; i < message.length(); i++) {
        char ch = message.charAt(i);
        System.out.println(ch);
    }
}
```

`char` versus `int`

Values of type `char` are stored internally as 16-bit integers. A standard encoding scheme called Unicode determines which integer value represents each character. (Unicode will be covered in more detail later in this chapter.) Since characters are

really integers, Java automatically converts a value of type `char` into an `int` whenever it is expecting an `int`:

```
char letter = 'a' + 2; // stores 'c'
```

It turns out that the integer value for `'a'` is `97`, so the expression's result is `99`, which is stored as the character `'c'`. An `int` can similarly be converted into a `char` using a type cast. (The cast is needed as a promise to the compiler, because not every possible `int` value corresponds to a valid character.) Below is an example of a code segment that uses a type cast to convert an `int` value to a value of type `char`:

```
int code = 66;
char grade = (char) code; // stores 'B'
```

Because values of type `char` are really integers, they can also be compared by using relational operators such as < or ==. In addition, they can be used in loops to cover ranges of letters. For example, the following code prints every letter of the alphabet:

```
for (char letter = 'a'; letter <= 'z'; letter++) {
    System.out.print(letter);
}
if (c == '8') {... // true
```

You can learn more about the character-to-integer equivalences by searching the web for Unicode tables.

Cumulative Text Algorithms

Strings of characters are often used in cumulative algorithms as discussed earlier in this chapter. For example, you might loop over the characters of a string searching for a particular letter. The following method accepts a string and a character and returns the number of times the character occurs in the string:

```
public static int count(String text, char c) {
    int found = 0;
    for (int i = 0; i < text.length(); i++) {
        if (text.charAt(i) == c) {
            found++;
        }
    }
    return found;
}
```

A `char` can be concatenated with a `String` using the standard + operator. Using this idea, a `String` can be built using a loop, starting with an empty string and

Table 4.5 Useful Methods of the `Character` Class

Method	Description	Example
`getNumericValue(ch)`	Converts a character that looks like a number into that number	`Character.getNumericValue('6')` returns 6
`isDigit(ch)`	Whether or not the character is one of the digits `'0'` through `'9'`	`Character.isDigit('X')` returns `false`
`isLetter(ch)`	Whether or not the character is in the range `'a'` to `'z'` or `'A'` to `'Z'`	`Character.isLetter('f')` returns `true`
`isLowerCase(ch)`	Whether or not the character is a lowercase letter	`Character.isLowerCase('Q')` returns `false`
`isUpperCase(ch)`	Whether or not the character is an uppercase letter	`Character.isUpperCase('Q')` returns `true`
`toLowerCase(ch)`	The lowercase version of the given letter	`Character.toLowerCase('Q')` returns `'q'`
`toUpperCase(ch)`	The uppercase version of the given letter	`Character.toUpperCase('x')` returns `'X'`

concatenating individual characters in the loop. This is called a *cumulative concatenation*. The following method accepts a string and returns the same characters in the reverse order:

```
public static String reverse(String phrase) {
    String result = "";
    for (int i = 0; i < phrase.length(); i++) {
        result = phrase.charAt(i) + result;
    }
    return result;
}
```

For example, the call of `reverse("Tin man")` returns `"nam niT"`.

Several useful methods can be called to check information about a character or convert one character into another. Remember that `char` is a primitive type, which means that you can't use the dot syntax used with `String`s. Instead, the methods are static methods in a class called `Character`; the methods accept `char` parameters and return appropriate values. Some of the most useful `Character` methods are listed in Table 4.5.

The following method counts the number of letters A–Z in a `String`, ignoring all nonletter characters such as punctuation, numbers, and spaces:

```
public static int countLetters(String phrase) {
    int count = 0;
    for (int i = 0; i < phrase.length(); i++) {
        char ch = phrase.charAt(i);
```

```
        if (Character.isLetter(ch)) {
            count++;
        }
    }
    return count;
}
```

For example, the call of `countLetters("gr8 JoB!")` returns 5.

System.out.printf

So far we've used `System.out.println` and `System.out.print` for console output. There's a third method, `System.out.printf`, which is a bit more complicated than the others but gives us some useful new abilities. The "f" in `printf` stands for "formatted," implying that `System.out.printf` gives you more control over the format in which your output is printed.

Imagine that you'd like to print a multiplication table from 1 to 10. The following code prints the correct numbers, but it doesn't look very nice:

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

The output is the following. Notice that the numbers don't line up vertically:

```
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

We could separate the numbers by tabs, which would be better. But this separation doesn't give us very much control over the appearance of the table. Every number would be exactly eight spaces apart on the screen, and the numbers would appear left-aligned. It would be a pain to try to right-align the numbers manually, because you'd have to use `if/else` statements to check whether a given number was in a certain range and, if necessary, pad it with a given number of spaces.

Did You Know?

ASCII and Unicode

We store data on a computer as binary numbers (sequences of 0s and 1s). To store textual data, we need an encoding scheme that will tell us what sequence of 0s and 1s to use for any given character. Think of it as a giant secret decoder ring that says things like, "If you want to store a lowercase 'a,' use the sequence 01100001."

In the early 1960s IBM developed an encoding scheme called *EBCDIC* that worked well with the company's punched cards, which had been in use for decades before computers were even invented. But it soon became clear that EBCDIC wasn't a convenient encoding scheme for computer programmers. There were gaps in the sequence that made characters like `'i'` and `'j'` appear far apart even though they follow one directly after the other.

In 1967 the American Standards Association published a scheme known as *ASCII* (pronounced "AS-kee") that has been in common use ever since. The acronym is short for "American Standard Code for Information Interchange." In its original form, ASCII defined 128 characters that each could be stored with 7 bits of data.

The biggest problem with ASCII is that it is an *American* code. There are many characters in common use in other countries that were not included in ASCII. For example, the British pound (£) and the Spanish variant of the letter n (ñ) are not included in the standard 128 ASCII characters. Various attempts have been made to extend ASCII, doubling it to 256 characters so that it can include many of these special characters. However, it turns out that even 256 characters is simply not enough to capture the incredible diversity of human communication.

Around the time that Java was created, a consortium of software professionals introduced a new standard for encoding characters known as *Unicode*. They decided that the 7 bits of standard ASCII and the 8 bits of extended ASCII were simply not big enough and chose not to set a limit on how many bits they might use for encoding characters. At the time of this writing, the consortium has identified over 110,000 characters, which require a little over 16 bits to store. Unicode includes the characters used in most modern languages and even some ancient languages. Egyptian hieroglyphs were added in 2007, although it still does not include Mayan hieroglyphs, and the consortium has rejected a proposal to include Klingon characters.

The designers of Java used Unicode as the standard for the type `char`, which means that Java programs are capable of manipulating a full range of characters. Fortunately, the Unicode Consortium decided to incorporate the ASCII encodings, so ASCII can be seen as a subset of Unicode. If you are curious about the actual ordering of characters in ASCII, type "ASCII table" into your favorite search engine and you will find millions of hits to explore.

A much easier way to print values aligned in fixed-width fields is to use the `System.out.printf` command. The `printf` method accepts a specially written `String` called a *format string* that specifies the general appearance of the output, followed by any parameters to be included in the output:

```
System.out.printf(<format string>, <parameter>, ..., <parameter>);
```

A format string is like a normal `String`, except that it can contain placeholders called *format specifiers* that allow you to specify a location where a variable's value should be inserted, along with the format you'd like to give that value. Format specifiers begin with a `%` sign and end with a letter specifying the kind of value, such as `d` for decimal integers (`int`) or `f` for floating-point numbers (real numbers of type `double`). Consider the following `printf` statement:

```
int x = 38, y = -152;
System.out.printf("location: (%d, %d)\n", x, y);
```

This statement produces the following output:

```
location: (38, -152)
```

The `%d` is not actually printed but is instead replaced with the corresponding parameter written after the format string. The number of format specifiers in the format string must match the number of parameters that follow it. The first specifier will be replaced by the first parameter, the second specifier by the second parameter, and so on. `System.out.printf` is unusual because it can accept a varying number of parameters.

The `printf` command is like `System.out.print` in that it doesn't move to a new line unless you explicitly tell it to do so. Notice that in the previous code we ended our format string with `\n` to complete the line of output.

Since a format specifier uses `%` as a special character, if you want to print an actual `%` sign in a `printf` statement, instead write two `%` characters in a row. For example:

```
int score = 87;
System.out.printf("You got %d%% on the exam!\n", score);
```

The code produces the following output:

```
You got 87% on the exam!
```

A format specifier can contain information after its `%` sign to specify the width, precision, and alignment of the value being printed. For example, `%8d` specifies an integer right-aligned in an 8-space-wide area, and `%12.4f` specifies a `double` value right-aligned in a 12-space-wide area, rounded to four digits past the decimal point. Table 4.6 lists some common format specifiers that you may wish to use in your programs.

Table 4.6 Common Format Specifiers

Specifier	Result
`%d`	Integer
`%8d`	Integer, right-aligned, 8-space-wide field
`%-6d`	Integer, left-aligned, 6-space-wide field
`%f`	Floating-point number
`%12f`	Floating-point number, right-aligned, 12-space-wide field
`%.2f`	Floating-point number, rounded to nearest hundredth
`%16.3f`	Floating-point number, rounded to nearest thousandth, 16-space-wide field
`%s`	String
`%8s`	String, right-aligned, 8-space-wide field
`%-9s`	String, left-aligned, 9-space-wide field

As a comprehensive example, suppose that the following variables have been declared to represent information about a student:

```
int score = 87;
double gpa = 3.18652;
String name = "Jessica";
```

The following code sample prints the preceding variables with several format specifiers:

```
System.out.printf("student name: %10s\n", name);
System.out.printf("exam score  : %10d\n", score);
System.out.printf("GPA         : %10.2f\n", gpa);
```

The code produces the following output:

```
student name:    Jessica
exam score  :         87
GPA         :       3.19
```

The three values line up on their right edge, because we print all of them with a width of 10. The `printf` method makes it easy to line up values in columns in this way. Notice that the student's GPA rounds to 3.19, because of the `2` in that variable's format specifier. The specifier `10.2` makes the value fit into an area 10 characters wide with exactly 2 digits after the decimal point.

Let's return to our multiplication table example. Now that we know about `printf`, we can print the table with right-aligned numbers relatively easily. We'll right-align the numbers into fields of width 5:

```
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10; j++) {
```

```
            System.out.printf("%5d", i * j);
        }
        System.out.println();
    }
```

This code produces the following output:

```
    1    2    3    4    5    6    7    8    9   10
    2    4    6    8   10   12   14   16   18   20
    3    6    9   12   15   18   21   24   27   30
    4    8   12   16   20   24   28   32   36   40
    5   10   15   20   25   30   35   40   45   50
    6   12   18   24   30   36   42   48   54   60
    7   14   21   28   35   42   49   56   63   70
    8   16   24   32   40   48   56   64   72   80
    9   18   27   36   45   54   63   72   81   90
   10   20   30   40   50   60   70   80   90  100
```

The `printf` method can also solve the problem with the `Roundoff` program introduced earlier in this chapter. Fixing the precision of the `double` value ensures that it will be rounded to avoid the tiny roundoff mistakes that result from `double` arithmetic. Here is the corrected program:

```
1 // Uses System.out.printf to correct roundoff errors.
2 public class Roundoff2 {
3     public static void main(String[] args) {
4         double n = 1.0;
5         for (int i = 1; i <= 10; i++) {
6             n += 0.1;
7             System.out.printf("%3.1f\n", n);
8         }
9     }
10 }
```

The program produces the following output:

```
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
2.0
```

4.4 Methods with Conditional Execution

We introduced a great deal of information about methods in Chapter 3, including how to use parameters to pass values into a method and how to use a `return` statement to have a method return a value. Now that we've introduced conditional execution, we need to revisit these issues so that you can gain a deeper understanding of them.

Preconditions and Postconditions

Every time you write a method you should think about exactly what that method is supposed to accomplish. You can describe how a method works by describing the *preconditions* that must be true before it executes and the *postconditions* that will be true after it has executed.

Precondition

A condition that must be true before a method executes in order to guarantee that the method can perform its task.

Postcondition

A condition that the method guarantees will be true after it finishes executing, as long as the preconditions were true before the method was called.

For example, if you are describing the task of a person on an automobile assembly line, you might use a postcondition like, "The bolts that secure the left front tire are on the car and tight." But postconditions are not the whole story. Employees on an assembly line depend on one another. A line worker can't add bolts and tighten them if the left tire isn't there or if there are no bolts. So, the assembly line worker might have preconditions like, "The left tire is mounted properly on the car, there are at least eight bolts in the supply box, and a working wrench is available." You describe the task fully by saying that the worker can make the postcondition(s) true if the precondition(s) are true before starting.

Like workers on an assembly line, methods need to work together, each solving its own portion of the task in order for them all to solve the overall task. The preconditions and postconditions describe the dependencies between methods.

Throwing Exceptions

We have seen several cases in which Java might throw an exception. For example, if we have a console `Scanner` and we call `nextInt`, the program will throw an exception if the user types something that isn't an `int`. In Appendix C we examine how you can handle exceptions. For now, we just want to explore some of the ways in which exceptions can occur and how you might want to generate them in your own code.

Ideally programs execute without generating any errors, but in practice various problems arise. If you ask the user for an integer, the user may accidentally or perhaps even maliciously type something that is not an integer. Or your code might have a bug in it.

The following program always throws an exception because it tries to compute the value of 1 divided by 0, which is mathematically undefined:

```
1 public class CauseException {
2     public static void main(String[] args) {
3         int x = 1 / 0;
4         System.out.println(x);
5     }
6 }
```

When you run the program, you get the following error message:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at CauseException.main(CauseException.java:3)
```

The problem occurs in line 3, when you ask Java to compute a value that can't be stored as an `int`. What is Java supposed to do with that value? It throws an exception that stops the program from executing and warns you that an arithmetic exception occurred while the program was executing that specific line of code.

It is worth noting that division by zero does not always produce an exception. You won't get an exception if you execute this line of code:

```
double x = 1.0 / 0.0;
```

In this case, the program executes normally and produces the output `Infinity`. This is because floating-point numbers follow a standard from the Institute of Electrical and Electronics Engineers (IEEE) that defines exactly what should happen in these cases, and there are special values representing infinity and `"NaN"` (not a number).

You may want to throw exceptions yourself in the code you write. In particular, it is a good idea to throw an exception if a precondition fails. For example, suppose that you want to write a method for computing the factorial of an integer. The factorial is defined as follows:

$n!$ (which is read as "n factorial") $= 1 * 2 * 3 * \ldots * n$

You can write a Java method that uses a cumulative product to compute this result:

```
public static int factorial(int n) {
    int product = 1;
    for (int i = 2; i <= n; i++) {
```

```
        product = product * i;
    }
    return product;
}
```

You can then test the method for various values with a loop:

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

The loop produces the following output:

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
```

It seems odd that the `factorial` method should return `1` when it is asked for 0!, but that is actually part of the mathematical definition of the factorial function. It returns `1` because the local variable `product` in the `factorial` method is initialized to `1`, and the loop is never entered when the parameter `n` has the value `0`. So, this is actually desirable behavior for 0!.

But what if you're asked to compute the factorial of a negative number? The method returns the same value, `1`. The mathematical definition of factorial says that the function is undefined for negative values of `n`, so it actually shouldn't even compute an answer when `n` is negative. Accepting only numbers that are zero or positive is a precondition of the method that can be described in the documentation:

```
// pre : n >= 0
// post: returns n factorial (n!)
```

Adding comments about this restriction is helpful, but what if someone calls the `factorial` method with a negative value anyway? The best solution is to throw an exception. The general syntax of the `throw` statement is:

```
throw <exception>;
```

In Java, exceptions are objects. Before you can throw an exception, you have to construct an exception object using new. You'll normally construct the object as you are throwing the exception, because the exception object includes information about what was going on when the error occurred. Java has a class called `IllegalArgumentException` that is meant to cover a case like this where someone has passed an inappropriate value as an argument. You can construct the exception object and include it in a `throw` statement as follows:

```
throw new IllegalArgumentException();
```

Of course, you'll want to do this only when the precondition fails, so you need to include the code inside an `if` statement:

```
if (n < 0) {
    throw new IllegalArgumentException();
}
```

You can also include some text when you construct the exception that will be displayed when the exception is thrown:

```
if (n < 0) {
    throw new IllegalArgumentException("negative n: " + n);
}
```

Incorporating the `pre/post` comments and the exception code into the method definition, you get the following code:

```
// pre : n >= 0
// post: returns n factorial (n!)
public static int factorial(int n) {
    if (n < 0) {
        throw new IllegalArgumentException("negative n: " + n);
    }
    int product = 1;
    for (int i = 2; i <= n; i++) {
        product = product * i;
    }
    return product;
}
```

You don't need an `else` after the `if` that throws the exception, because when an exception is thrown, it halts the execution of the method. So, if someone calls the `factorial` method with a negative value of `n`, Java will never execute the code that follows the `throw` statement.

You can test this code with the following `main` method:

```
public static void main(String[] args) {
    System.out.println(factorial(-1));
}
```

When you execute this program, it stops executing and prints the following message:

```
Exception in thread "main"
java.lang.IllegalArgumentException: negative n: -1
        at Factorial2.factorial(Factorial2.java:8)
        at Factorial2.main(Factorial2.java:3)
```

The message indicates that the program `Factorial2` stopped running because an `IllegalArgumentException` was thrown with a negative `n` of `-1`. The system then shows you a backward trace of how it got there. The illegal argument appeared in line 8 of the `factorial` method of the `Factorial2` class. It got there because of a call in line 3 of the `main` of the `Factorial2` class. This kind of information is very helpful when you want to find the bugs in your programs.

Throwing exceptions is an example of *defensive programming*. We don't intend to have bugs in the programs we write, but we're only human, so we want to build in mechanisms that will give us feedback when we make mistakes. Writing code that will test the values passed to methods and throw an `IllegalArgumentException` when a value is not appropriate is a great way to provide that feedback.

Revisiting Return Values

In Chapter 3 we looked at some examples of simple calculating methods that return a value, as in this method for finding the sum of the first *n* integers:

```
public static int sum(int n) {
    return (n + 1) * n / 2;
}
```

Now that you know how to write `if/else` statements, we can look at some more interesting examples involving return values. For example, earlier in this chapter you saw that the `Math` class has a method called `max` that returns the larger of two values. There are actually two different versions of the method, one that finds the larger of two integers and one that finds the larger of two `doubles`. Recall that when two methods have the same name (but different parameters), it is called overloading.

Let's write our own version of the `max` method that returns the larger of two integers. Its header will look like this:

```
public static int max(int x, int y) {
    ...
}
```

We want to return either `x` or `y`, depending on which is larger. This is a perfect place to use an `if/else` construct:

```
public static int max(int x, int y) {
    if (x > y) {
        return x;
    } else {
        return y;
    }
}
```

This code begins by testing whether `x` is greater than `y`. If it is, the computer executes the first branch by returning `x`. If it is not, the computer executes the `else` branch by returning `y`. But what if `x` and `y` are equal? The preceding code executes the `else` branch when the values are equal, but it doesn't actually matter which `return` statement is executed when `x` and `y` are equal.

Remember that when Java executes a `return` statement, the method stops executing. It's like a command to Java to "get out of this method right now." That means that this method could also be written as follows:

```
public static int max(int x, int y) {
    if (x > y) {
        return x;
    }
    return y;
}
```

This version of the code is equivalent in behavior because the statement `return x` inside the `if` statement will cause Java to exit the method immediately and Java will not execute the `return` statement that follows the `if`. On the other hand, if we don't enter the `if` statement, we proceed directly to the statement that follows it (`return y`).

Whether you choose to use the first form or the second in your own programs depends somewhat on personal taste. The `if/else` construct makes it more clear that the method is choosing between two alternatives, but some people prefer the second alternative because it is shorter.

As another example, consider the `indexOf` method of the `String` class. We'll define a variable `s` that stores the following `String`:

```
String s = "four score and seven years ago";
```

Now we can write expressions like the following to determine where a particular character appears in the `String`:

```
int r = s.indexOf('r');
int v = s.indexOf('v');
```

This code sets `r` to `3` because 3 is the index of the first occurrence of the letter `'r'` in the `String`. It sets `v` to `17` because that is the index of the first occurrence of the letter `'v'` in the `String`.

The `indexOf` method is part of the `String` class, but let's see how we could write a different method that performs the same task. Our method would be called differently because it is a static method outside the `String` object. We would have to pass it both the `String` and the letter:

```
int r = indexOf('r', s);
int v = indexOf('v', s);
```

So, the header for our method would be:

```
public static int indexOf(char ch, String s) {
   ...
}
```

Remember that when a method returns a value, we must include the return type after the words `public static`. In this case, we have indicated that the method returns an `int` because the index will be an integer.

This task can be solved rather nicely with a `for` loop that goes through each possible index from first to last. We can describe this in pseudocode as follows:

```
for (each index i in the string) {
    if the char is at position i, we've found it.
}
```

To flesh this out, we have to think about how to test whether the character at position `i` is the one we are looking for. Remember that `String` objects have a method called `charAt` that allows us to pull out an individual character from the `String`, so we can refine our pseudocode as follows:

```
for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == ch) {
      we've found it.
    }
}
```

To complete this code, we have to refine what to do when "we've found it." If we find the character, we have our answer: the current value of the variable `i`. And if that is the answer we want to return, we can put a `return` statement there:

```
for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == ch) {
        return i;
    }
}
```

To understand this code, you have to understand how the `return` statement works. For example, if the `String` `s` is the one from our example ("four score...") and we are searching for the character `'r'`, we know that when `i` is equal to `3` we will find that `s.charAt(3)` is equal to `'r'`. That case causes our code to execute the `return` statement, effectively saying:

```
return 3;
```

When a `return` statement is executed, Java immediately exits the method, which means that we break out of the loop and return `3` as our answer. Even though the loop would normally increment `i` to `4` and keep going, our code doesn't do that because we hit the `return` statement.

There is only one thing missing from our code. If we try to compile it as it is, we get this error message from the Java compiler:

```
missing return statement
```

This error message occurs because we haven't told Java what to do if we never find the character we are searching for. In that case, we will execute the `for` loop in its entirety and reach the end of the method without having returned a value. This is not acceptable. If we say that the method returns an `int`, we have to guarantee that every path through the method will return an `int`.

If we don't find the character, we want to return some kind of special value to indicate that the character was not found. We can't use the value `0`, because `0` is a legal index for a `String` (the index of the first character). So, the convention in Java is to return `-1` if the character is not found. It is easy to add the code for this `return` statement after the `for` loop:

```
public static int indexOf(char ch, String s) {
    for (int i = 0; i < s.length(); i++) {
        if (s.charAt(i) == ch) {
            return i;
        }
    }
    return -1;
}
```

Common Programming Error

String Index Out of Bounds

It's very easy to forget that the last index of a `String` of length n is actually $n - 1$. Forgetting this fact can cause you to write incorrect text-processing loops like this one:

```
// This version of the code has a mistake!
// The test should be i < s.length()
public static int indexOf(char ch, String s) {
    for (int i = 0; i <= s.length(); i++) {
        if (s.charAt(i) == ch) {
            return i;
        }
    }
    return -1;
}
```

The program will throw an exception if the loop runs past the end of the `String`. On the last pass through the loop, the value of the variable `i` will be equal to `s.length()`. When it executes the `if` statement test, the program will throw the exception. The error message will resemble the following:

```
Exception in thread "main"
  java.lang.StringIndexOutOfBoundsException:
  String index out of range: 11
      at java.lang.String.charAt(Unknown Source)
      at OutOfBoundsExample.indexOf(OutOfBoundsExample.java:9)
      at OutOfBoundsExample.main(OutOfBoundsExample.java:4)
```

An interesting thing about the bug in this example is that it only occurs if the `String` does not contain the character `ch`. If `ch` is contained in the `String`, the `if` test will be `true` for one of the legal indexes in `s`, so the code will return that index. Only if all the characters from `s` have been examined without finding `ch` will the loop attempt its last fatal pass.

It may seem strange that we don't have a test for the final `return` statement that returns `-1`, but remember that the `for` loop tries every possible index of the `String` searching for the character. If the character appears anywhere in the `String`, the `return` statement inside the loop will be executed and we'll never get to the `return` statement after the loop. The only way to get to the `return` statement after the loop is to find that the character appears nowhere in the given `String`.

Reasoning about Paths

The combination of `if/else` and `return` is powerful. It allows you to solve many complex problems in the form of a method that accepts some input and computes a result. But you have to be careful to think about the different paths that exist in the code that you write. At first this process might seem annoying, but when you get the hang of it, you will find that it allows you to simplify your code.

For example, suppose that we want to convert scores on the SAT into a rating to be used for college admission. Each of the three components of the SAT ranges from 200 to 800, so the overall total ranges from 600 to 2400. Suppose that a hypothetical college breaks up this range into three subranges with totals below 1200 considered not competitive, scores of at least 1200 but less than 1800 considered competitive, and scores of 1800 to 2400 considered highly competitive.

Let's write a method called `rating` that will take the total SAT score as a parameter and will return a string with the appropriate text. We can use the AND operator described earlier to write an `if/else` construct that has tests for each of these ranges:

```
public static String rating(int totalSAT) {
    if (totalSAT >= 600 && totalSAT < 1200) {
        return "not competitive";
    } else if (totalSAT >= 1200 && totalSAT < 1800) {
        return "competitive";
    } else if (totalSAT >= 1800 && totalSAT <= 2400) {
        return "highly competitive";
    }
}
```

This method has been written in a logical manner with specific tests for each of the three cases, but it doesn't compile. The compiler indicates at the end of the method that there was a "missing return statement." That seems odd because there are three different `return` statements in this method. We have included a `return` for each of the different cases, so why is there a compiler error?

When the compiler encounters a method that is supposed to return a value, it computes every possible path through the method and makes sure that each path ends with a call on `return`. The method we have written has four paths through it. If the first test succeeds, then the method returns `"not competitive"`. Otherwise, if the second test succeeds, then the method returns `"competitive"`. If both of those tests fail but the third test succeeds, then the method returns `"highly competitive"`. But what if all three tests fail? That case would constitute a fourth path that doesn't have a `return` statement associated with it. Instead, we would reach the end of the method without having returned a value. That is not acceptable, which is why the compiler produces an error message.

It seems annoying that we have to deal with a fourth case because we know that the total SAT score will always be in the range of 600 to 2400. Our code covers all of

the cases that we expect for this method, but that isn't good enough. Java insists that we cover every possible case.

Understanding this idea can simplify the code you write. If you think in terms of paths and cases, you can often eliminate unnecessary code. For our method, if we really want to return just one of three different values, then we don't need a third test. We can make the final branch of the nested `if/else` be a simple `else`:

```
public static String rating(int totalSAT) {
    if (totalSAT >= 600 && totalSAT < 1200) {
        return "not competitive";
    } else if (totalSAT >= 1200 && totalSAT < 1800) {
        return "competitive";
    } else {  // totalSAT >= 1800
        return "highly competitive";
    }
}
```

This version of the method compiles and returns the appropriate string for each different case. We were able to eliminate the final test because we know that we want only three paths through the method. Once we have specified two of the paths, then everything else must be part of the third path.

We can carry this idea one step further. We've written a method that compiles and computes the right answer, but we can make it even simpler. Consider the first test, for example. Why should we test for the total being greater than or equal to 600? If we expect that it will always be in the range of 600 to 2400, then we can simply test whether the total is less than 1200. Similarly, to test for the highly competitive range, we can simply test whether the score is at least 1800. Of the three ranges, these are the two simplest to test for. So we can simplify this method even further by including tests for the first and third subranges and assume that all other totals are in the middle range:

```
public static String rating(int totalSAT) {
    if (totalSAT < 1200) {
        return "not competitive";
    } else if (totalSAT >= 1800) {
        return "highly competitive";
    } else {  // 1200 <= totalSAT < 1800
        return "competitive";
    }
}
```

Whenever you write a method like this, you should think about the different cases and figure out which ones are the simplest to test for. This will allow you to avoid writing an explicit test for the most complex case. As in these examples, it is a good idea to include a comment on the final `else` branch to describe that particular case in English.

Before we leave this example, it is worth thinking about what happens when the method is passed an illegal SAT total. If it is passed a total less than 600, then it classifies it as not competitive and if it passed a total greater than 2400, it will classify it as highly competitive. Those aren't bad answers for the program to give, but the right thing to do is to document the fact that there is a precondition on the total. In addition, we can add an extra test for this particular case and throw an exception if the precondition is violated. Testing for the illegal values is a case in which the logical OR is appropriate because illegal values will either be too low or too high (but not both):

```
// pre: 600 <= totalSAT <= 2400 (throws IllegalArgumentException if not)
public static String rating(int totalSAT) {
    if (totalSAT < 600 || totalSAT > 2400) {
        throw new IllegalArgumentException("total: " + totalSAT);
    } else if (totalSAT < 1200) {
        return "not competitive";
    } else if (totalSAT >= 1800) {
        return "highly competitive";
    } else {  // 1200 <= totalSAT < 1800
        return "competitive";
    }
}
```

4.5 Case Study: Body Mass Index

Individual body mass index has become a popular measure of overall health. The Centers for Disease Control and Prevention (CDC) website about body mass index (http://www.cdc.gov/healthyweight/assessing/bmi/index.html) explains:

> Body Mass Index (BMI) is a number calculated from a person's weight and height. BMI provides a reliable indicator of body fatness for most people and is used to screen for weight categories that may lead to health problems.

It has also become popular to compare the statistics for two or more individuals who are pitted against one another in a "fitness challenge," or to compare two sets of numbers for the same person to get a sense of how that person's BMI will vary if a person loses weight. In this section, we will write a program that prompts the user for the height and weight of two individuals and reports the overall results for the two people. Here is a sample execution for the program we want to write:

```
This program reads data for two
people and computes their body
mass index and weight status.
```

```
Enter next person's information:
height (in inches)? 73.5
weight (in pounds)? 230

Enter next person's information:
height (in inches)? 71
weight (in pounds)? 220.5

Person #1 body mass index = 29.93
overweight
Person #2 body mass index = 30.75
obese
```

In Chapter 1 we introduced the idea of *iterative enhancement,* in which you develop a complex program in stages. Every professional programmer uses this technique, so it is important to learn to apply it yourself in the programs you write.

In this case, we eventually want our program to explain to the user what it does and compute BMI results for two different people. We also want the program to be well structured. But we don't have to do everything at once. In fact, if we try to do so, we are likely to be overwhelmed by the details. In writing this program, we will go through three different stages:

1. First, we'll write a program that computes results for just one person, without an introduction. We won't worry about program structure yet.
2. Next, we'll write a complete program that computes results for two people, including an introduction. Again, we won't worry about program structure at this point.
3. Finally, we will put together a well-structured and complete program.

One-Person Unstructured Solution

Even the first version of the program will prompt for user input, so we will need to construct a `Scanner` object to read from the console:

```
Scanner console = new Scanner(System.in);
```

To compute the BMI for an individual, we will need to know the height and weight of that person. This is a fairly straightforward "prompt and read" task. The only real decision here is with regard to the type of variable to use for storing the height and weight. People often talk about height and weight in whole numbers, but the question to ask is whether or not it makes sense for people to use fractions. Do people ever describe their heights using half-inches? The answer is yes. Do people ever describe their weights using half-pounds? Again the answer is yes. So it makes sense to store the values as doubles, to allow people to enter either integer values or fractions:

```
System.out.println("Enter next person's information:");
System.out.print("height (in inches)? ");
```

```
double height1 = console.nextDouble();
System.out.print("weight (in pounds)? ");
double weight1 = console.nextDouble();
```

Once we have the person's height and weight, we can compute the person's BMI. The CDC website gives the following BMI formula for adults:

$$\frac{\text{weight (lb)}}{[\text{height (in)}]^2} \times 703$$

This formula is fairly easy to translate into a Java expression:

```
double bmi1 = weight1 / (height1 * height1) * 703;
```

If you look closely at the sample execution, you will see that we want to print blank lines to separate different parts of the user interaction. The introduction ends with a blank line, then there is a blank line after the "prompt and read" portion of the interaction. So, after we add an empty `println` and put all of these pieces together, our `main` method looks like this:

```
public static void main(String[] args) {
    Scanner console = new Scanner(System.in);
    System.out.println("Enter next person's information:");
    System.out.print("height (in inches)? ");
    double height1 = console.nextDouble();
    System.out.print("weight (in pounds)? ");
    double weight1 = console.nextDouble();
    double bmi1 = weight1 / (height1 * height1) * 703;
    System.out.println();
    ...
}
```

This program prompts for values and computes the BMI. Now we need to include code to report the results. We could use a `println` for the BMI:

```
System.out.println("Person #1 body mass index = " + bmi1);
```

This would work, but it produces output like the following:

```
Person #1 body mass index = 29.930121708547368
```

The long sequence of digits after the decimal point is distracting and implies a level of precision that we simply don't have. It is more appropriate and more appealing to the user to list just a few digits after the decimal point. This is a good place to use a `printf`:

```
System.out.printf("Person #1 body mass index = %5.2f\n", bmi1);
```

Table 4.7 Weight Status by BMI

BMI	Weight status
below 18.5	underweight
18.5–24.9	normal
25.0–29.9	overweight
30.0 and above	obese

In the sample execution we also see a report of the person's weight status. The CDC website includes the information shown in Table 4.7. There are four entries in this table, so we need four different `println` statements for the four possibilities. We will want to use `if` or `if/else` statements to control the four `println` statements. In this case, we know that we want to print exactly one of the four possibilities. Therefore, it makes most sense to use a nested `if/else` construct that ends with an `else`.

But what tests do we use for the nested `if/else`? If you look closely at Table 4.7, you will see that there are some gaps. For example, what if your BMI is 24.95? That number isn't between 18.5 and 24.9 and it isn't between 25.0 and 29.9. It seems clear that the CDC intended its table to be interpreted slightly differently. The range is probably supposed to be 18.5–24.999999 (repeating), but that would look rather odd in a table. In fact, if you understand nested `if/else` statements, this is a case in which a nested `if/else` construct expresses the possibilities more clearly than a table like the CDC's. The nested `if/else` construct looks like this:

```
if (bmi1 < 18.5) {
    System.out.println("underweight");
} else if (bmi1 < 25) {
    System.out.println("normal");
} else if (bmi1 < 30) {
    System.out.println("overweight");
} else {  // bmi1 >= 30
    System.out.println("obese");
}
```

So, putting all this together, we get a complete version of the first program:

```
1  import java.util.*;
2
3  public class BMI1 {
4      public static void main(String[] args) {
5          Scanner console = new Scanner(System.in);
6
7          System.out.println("Enter next person's information:");
8          System.out.print("height (in inches)? ");
```

```
 9          double height1 = console.nextDouble();
10          System.out.print("weight (in pounds)? ");
11          double weight1 = console.nextDouble();
12          double bmi1 = weight1 / (height1 * height1) * 703;
13          System.out.println();
14
15          System.out.printf("Person #1 body mass index = %5.2f\n", bmi1);
16          if (bmi1 < 18.5) {
17              System.out.println("underweight");
18          } else if (bmi1 < 25) {
19              System.out.println("normal");
20          } else if (bmi1 < 30) {
21              System.out.println("overweight");
22          } else {  // bmi1 >= 30
23              System.out.println("obese");
24          }
25      }
26  }
```

Here is a sample execution of the program:

```
Enter next person's information:
height (in inches)? 73.5
weight (in pounds)? 230

Person #1 body mass index = 29.93
overweight
```

Two-Person Unstructured Solution

Now that we have a program that computes one person's BMI and weight status, let's expand it to handle two different people. Experienced programmers would probably begin by adding structure to the program before trying to make it handle two sets of data, but novice programmers will find it easier to consider the unstructured solution first.

To make this program handle two people, we can copy and paste a lot of the code and make slight modifications. For example, instead of using variables called `height1`, `weight1`, and `bmi1`, for the second person we will use variables `height2`, `weight2`, and `bmi2`.

We also have to be careful to do each step in the right order. Looking at the sample execution, you'll see that the program prompts for data for both individuals first and then reports results for both. Thus, we can't copy the entire program and simply paste a second copy; we have to rearrange the order of the statements so that all of the prompting happens first and all of the reporting happens later.

We've also decided that when we move to this second stage, we will add code for the introduction. This code should appear at the beginning of the program and should include an empty `println` to produce a blank line to separate the introduction from the rest of the user interaction.

We now combine these elements into a complete program:

```
1  // This program finds the body mass index (BMI) for two
2  // individuals.
3
4  import java.util.*;
5
6  public class BMI2 {
7      public static void main(String[] args) {
8          System.out.println("This program reads data for two");
9          System.out.println("people and computes their body");
10         System.out.println("mass index and weight status.");
11         System.out.println();
12
13         Scanner console = new Scanner(System.in);
14
15         System.out.println("Enter next person's information:");
16         System.out.print("height (in inches)? ");
17         double height1 = console.nextDouble();
18         System.out.print("weight (in pounds)? ");
19         double weight1 = console.nextDouble();
20         double bmi1 = weight1 / (height1 * height1) * 703;
21         System.out.println();
22
23         System.out.println("Enter next person's information:");
24         System.out.print("height (in inches)? ");
25         double height2 = console.nextDouble();
26         System.out.print("weight (in pounds)? ");
27         double weight2 = console.nextDouble();
28         double bmi2 = weight2 / (height2 * height2) * 703;
29         System.out.println();
30
31         System.out.printf("Person #1 body mass index = %5.2f\n", bmi1);
32         if (bmi1 < 18.5) {
33             System.out.println("underweight");
34         } else if (bmi1 < 25) {
35             System.out.println("normal");
36         } else if (bmi1 < 30) {
37             System.out.println("overweight");
38         } else {  // bmi1 >= 30
```

```
39              System.out.println("obese");
40          }
41
42          System.out.printf("Person #2 body mass index = %5.2f\n", bmi2);
43          if (bmi2 < 18.5) {
44              System.out.println("underweight");
45          } else if (bmi2 < 25) {
46              System.out.println("normal");
47          } else if (bmi2 < 30) {
48              System.out.println("overweight");
49          } else {  // bmi2 >= 30
50              System.out.println("obese");
51          }
52      }
53  }
```

This program compiles and works. When we execute it, we get exactly the interaction we wanted. However, the program lacks structure. All of the code appears in `main`, and there is significant redundancy. That shouldn't be a surprise, because we created this version by copying and pasting. Whenever you find yourself using copy and paste, you should wonder whether there isn't a better way to solve the problem. Usually there is.

Two-Person Structured Solution

Let's explore how static methods can improve the structure of the program. Looking at the code, you will notice a great deal of redundancy. For example, we have two code segments that look like this:

```
System.out.println("Enter next person's information:");
System.out.print("height (in inches)? ");
double height1 = console.nextDouble();
System.out.print("weight (in pounds)? ");
double weight1 = console.nextDouble();
double bmi1 = weight1 / (height1 * height1) * 703;
System.out.println();
```

The only difference between these two code segments is that the first uses variables `height1`, `weight1`, `bmi1`, and the second uses variables `height2`, `weight2`, and `bmi2`. We eliminate redundancy by moving code like this into a method that we can call twice. So, as a first approximation, we can turn this code into a more generic form as the following method:

```
public static void getBMI(Scanner console) {
    System.out.println("Enter next person's information:");
    System.out.print("height (in inches)? ");
```

```
    double height = console.nextDouble();
    System.out.print("weight (in pounds)? ");
    double weight = console.nextDouble();
    double bmi = weight / (height * height) * 703;
    System.out.println();
}
```

We have to pass in the `Scanner` from `main`. Otherwise we have made all the variables local to this method. From `main` we can call this method twice:

```
getBMI(console);
getBMI(console);
```

Unfortunately, introducing this change breaks the rest of the code. If we try to compile and run the program, we find that we get error messages in `main` whenever we refer to the variables `bmi1` and `bmi2`.

The problem is that the method computes a `bmi` value that we need later in the program. We can fix this by having the method return the `bmi` value that it computes:

```
public static double getBMI(Scanner console) {
    System.out.println("Enter next person's information:");
    System.out.print("height (in inches)? ");
    double height = console.nextDouble();
    System.out.print("weight (in pounds)? ");
    double weight = console.nextDouble();
    double bmi = weight / (height * height) * 703;
    System.out.println();
    return bmi;
}
```

Notice that the method header now lists the return type as `double`. We also have to change `main`. We can't just call the method twice the way we would call a `void` method. Because each call returns a BMI result that the program will need later, for each call we have to store the result coming back from the method in a variable:

```
double bmi1 = getBMI(console);
double bmi2 = getBMI(console);
```

Study this change carefully, because this technique can be one of the most challenging for novices to master. When we write the method, we have to make sure that it returns the BMI result. When we write the call, we have to make sure that we store the result in a variable so that we can access it later.

After this modification, the program will compile and run properly. But there is another obvious redundancy in the `main` method: The same nested `if/else` construct appears twice. The only difference between them is that in one case we use the

variable `bmi1`, and in the other case we use the variable `bmi2`. The construct is easily generalized with a parameter:

```
public static void reportStatus(double bmi) {
    if (bmi < 18.5) {
        System.out.println("underweight");
    } else if (bmi < 25) {
        System.out.println("normal");
    } else if (bmi < 30) {
        System.out.println("overweight");
    } else {  // bmi >= 30
        System.out.println("obese");
    }
}
```

Using this method, we can replace the code in `main` with two calls:

```
System.out.printf("Person #1 body mass index = %5.2f\n", bmi1);
reportStatus(bmi1);
System.out.printf("Person #2 body mass index = %5.2f\n", bmi2);
reportStatus(bmi2);
```

That change takes care of the redundancy in the program, but we can still use static methods to improve the program by better indicating structure. It is best to keep the `main` method short if possible, to reflect the overall structure of the program. The problem breaks down into three major phases: introduction, the computation of the BMI, and the reporting of the results. We already have a method for computing the BMI, but we haven't yet introduced methods for the introduction and reporting of results. It is fairly simple to add these methods.

There is one other method that we should add to the program. We are using a formula from the CDC website for calculating the BMI of an individual given the person's height and weight. Whenever you find yourself programming a formula, it is a good idea to introduce a method for that formula so that it is easy to spot and so that it has a name.

Applying all these ideas, we end up with the following version of the program:

```
1  // This program finds the body mass index (BMI) for two
2  // individuals. This variation includes several methods
3  // other than main.
4
5  import java.util.*;
6
7  public class BMI3 {
8      public static void main(String[] args) {
9          giveIntro();
```

```
10          Scanner console = new Scanner(System.in);
11          double bmi1 = getBMI(console);
12          double bmi2 = getBMI(console);
13          reportResults(bmi1, bmi2);
14      }
15
16      // introduces the program to the user
17      public static void giveIntro() {
18          System.out.println("This program reads data for two");
19          System.out.println("people and computes their body");
20          System.out.println("mass index and weight status.");
21          System.out.println();
22      }
23
24      // prompts for one person's statistics, returning the BMI
25      public static double getBMI(Scanner console) {
26          System.out.println("Enter next person's information:");
27          System.out.print("height (in inches)? ");
28          double height = console.nextDouble();
29          System.out.print("weight (in pounds)? ");
30          double weight = console.nextDouble();
31          double bmi = BMIFor(height, weight);
32          System.out.println();
33          return bmi;
34      }
35
36      // this method contains the body mass index formula for
37      // converting the given height (in inches) and weight
38      // (in pounds) into a BMI
39      public static double BMIFor(double height, double weight) {
40          return weight / (height * height) * 703;
41      }
42
43      // reports the overall bmi values and weight status
44      public static void reportResults(double bmi1, double bmi2) {
45          System.out.printf("Person #1 body mass index = %5.2f\n", bmi1);
46          reportStatus(bmi1);
47          System.out.printf("Person #2 body mass index = %5.2f\n",bmi2);
48          reportStatus(bmi2);
49      }
50
51      // reports the weight status for the given BMI value
52      public static void reportStatus(double bmi) {
53          if (bmi < 18.5) {
```

```
54              System.out.println("underweight");
55          } else if (bmi < 25) {
56              System.out.println("normal");
57          } else if (bmi < 30) {
58              System.out.println("overweight");
59          } else {  // bmi >= 30
60              System.out.println("obese");
61          }
62      }
63  }
```

This solution interacts with the user the same way and produces the same results as the unstructured solution, but it has a much nicer structure. The unstructured program is in a sense simpler, but the structured solution is easier to maintain if we want to expand the program or make other modifications. These structural benefits aren't so important in short programs, but they become essential as programs become longer and more complex.

Procedural Design Heuristics

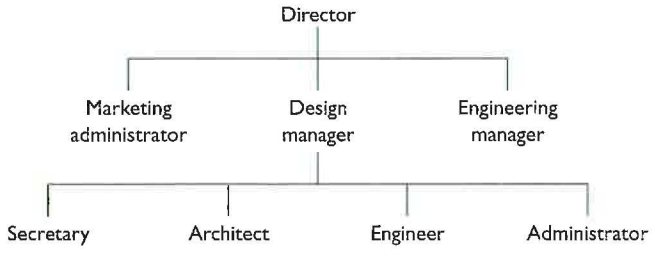
There are often many ways to divide (decompose) a problem into methods, but some sets of methods are better than others. Decomposition is often vague and challenging, especially for larger programs that have complex behavior. But the rewards are worth the effort, because a well-designed program is more understandable and more modular. These features are important when programmers work together or when revisiting a program written earlier to add new behavior or modify existing code. There is no single perfect design, but in this section we will discuss several *heuristics* (guiding principles) for effectively decomposing large programs into methods.

Consider the following alternative poorly structured implementation of the single-person BMI program. We'll use this program as a counterexample, highlighting places where it violates our heuristics and giving reasons that it is worse than the previous complete version of the BMI program.

```
1  // A poorly designed version of the BMI case study program.
2
3  import java.util.*;
4
5  public class BadBMI {
6      public static void main(String[] args) {
7          System.out.println("This program reads data for one");
8          System.out.println("person and computes his/her body");
9          System.out.println("mass index and weight status.");
10         System.out.println();
11
12         Scanner console = new Scanner(System.in);
13         person(console);
```

```
14      }
15
16      public static void person(Scanner console) {
17          System.out.println("Enter next person's information:");
18          System.out.print("height (in inches)? ");
19          double height = console.nextDouble();
20          getWeight(console, height);
21      }
22
23      public static void getWeight(Scanner console, double height) {
24          System.out.print("weight (in pounds)? ");
25          double weight = console.nextDouble();
26          reportStatus(console, height, weight);
27      }
28
29      public static void reportStatus(Scanner console, double height,
30                                      double weight) {
31          double bmi = weight / (height * height) * 703;
32          System.out.println("Person #1 body mass index = " + bmi);
33          if (bmi < 18.5) {
34              System.out.println("underweight");
35          } else if (bmi < 25) {
36              System.out.println("normal");
37          } else if (bmi < 30) {
38              System.out.println("overweight");
39          } else {
40              System.out.println("obese");
41          }
42      }
43  }
```

The methods of a program are like workers in a company. The author of a program acts like the director of a company, deciding what employee positions to create, how to group employees together into working units, which work to task to which group, and how groups will interact. Suppose a company director were to divide work into three major departments, two of which are overseen by middle managers:



A good structure gives each group clear tasks to complete, avoids giving any particular person or group too much work, and provides a balance between workers and management. These guidelines lead to the first of our procedural design heuristics.

1. Each method should have a coherent set of responsibilities. In our analogy to a company, each group of employees must have a clear idea of what work it is to perform. If any of the groups does not have clear responsibilities, it's difficult for the company director to keep track of who is working on what task. When a new job comes in, two departments might both try to claim it, or a job might go unclaimed by any department.

The analogous concept in programming is that each method should have a clear purpose and set of responsibilities. This characteristic of computer programs is called *cohesion.*

Cohesion

A desirable quality in which the responsibilities of a method or process are closely related to each other.

A good rule of thumb is that you should be able to summarize each of your methods in a single sentence such as "The purpose of this method is to ... ." Writing a sentence like this is a good way to develop a comment for a method's header. It's a bad sign when you have trouble describing the method in a single sentence or when the sentence is long and uses the word "and" several times. Those indications can mean that the method is too large, too small, or does not perform a cohesive set of tasks.

The methods of the `BadBMI` example have poor cohesion. The `person` method's purpose is vague, and `getWeight` is probably too trivial to be its own method. The `reportStatus` method would be more readable if the computation of the BMI were its own method, since the formula is complex.

A subtler application of this first heuristic is that not every method must produce output. Sometimes a method is more reusable if it simply computes a complex result and returns it rather than printing the result that was computed. This format leaves the caller free to choose whether to print the result or to use it to perform further computations. In the `BadBMI` program, the `reportStatus` method both computes and prints the user's BMI. The program would be more flexible if it had a method to simply compute and return the BMI value, such as `BMIFor` in the `BMI3` version of the code. Such a method might seem trivial because its body is just one line in length, but it has a clear, cohesive purpose: capturing a complex expression that is used several times in the program.

2. No one method should do too large a share of the overall task. One subdivision of a company cannot be expected to design and build the entire product line for the year. This system would overwork that subdivision and would leave the other divisions without enough work to do. It would also make it difficult for the subdivisions to communicate effectively, since so much important information and responsibility would be concentrated among so few people.

Similarly, one method should not be expected to comprise the bulk of a program. This principle follows naturally from our first heuristic regarding cohesion, because a method that does too much cannot be cohesive. We sometimes refer to methods like these as "do-everything" methods because they do nearly everything involved in solving the problem. You may have written a "do-everything" method if one of your methods is much longer than the others, hoards most of the variables and data, or contains the majority of the logic and loops.

In the `BadBMI` program, the `person` method is an example of a do-everything method. This fact may seem surprising, since the method is not very many lines long. But a single call to `person` leads to several other calls that collectively end up doing all of the work for the program.

3. Coupling and dependencies between methods should be minimized. A company is more productive if each of its subdivisions can largely operate independently when completing small work tasks. Subdivisions of the company do need to communicate and depend on each other, but such communication comes at a cost. Interdepartmental interactions are often minimized and kept to meetings at specific times and places.

When we are programming, we try to avoid methods that have tight *coupling*.

> **Coupling**
>
> An undesirable state in which two methods or processes rigidly depend on each other.

Methods are coupled if one cannot easily be called without the other. One way to determine how tightly coupled two methods are is to look at the set of parameters one passes to the other. A method should accept a parameter only if that piece of data needs to be provided from outside and only if that data is necessary to complete the method's task. In other words, if a piece of data could be computed or gathered inside the method, or if the data isn't used by the method, it should not be declared as a parameter to the method.

An important way to reduce coupling between methods is to use `return` statements to send information back to the caller. A method should return a result value if it computes something that may be useful to later parts of the program. Because it is desirable for methods to be cohesive and self-contained, it is often better for the program to return a result than to call further methods and pass the result as a parameter to them.

None of the methods in the `BadBMI` program returns a value. Each method passes parameters to the next methods, but none of them returns the value. This is a lost opportunity because several values (such as the user's height, weight, or BMI) would be better handled as `return` values.

4. The `main` method should be a concise summary of the overall program. The top person in each major group or department of our hypothetical company reports to the group's director. If you look at the groups that are directly connected to the director at the top level of the company diagram, you can see a summary of the overall work: design, engineering, and marketing. This structure helps the director stay aware of what each group is doing. Looking at the top-level structure can also help the employees get a quick overview of the company's goals.

A program's `main` method is like the director in that it begins the overall task and executes the various subtasks. A `main` method should read as a summary of the overall program's behavior. Programmers can understand each other's code by looking at `main` to get a sense of what the program is doing as a whole.

A common mistake that prevents `main` from being a good program summary is the inclusion of a "do-everything" method. When the `main` method calls it, the do-everything method proceeds to do most or all of the real work.

Another mistake is setting up a program in such a way that it suffers from *chaining*.

Chaining

An undesirable design in which a "chain" of several methods call each other without returning the overall flow of control to `main`.

A program suffers from chaining if the end of each method simply calls the next method. Chaining often occurs when a new programmer does not fully understand returns and tries to avoid using them by passing more and more parameters down to the rest of the program. Figure 4.8 shows a hypothetical program with two designs. The flow of calls in a badly chained program might look like the diagram on the left.

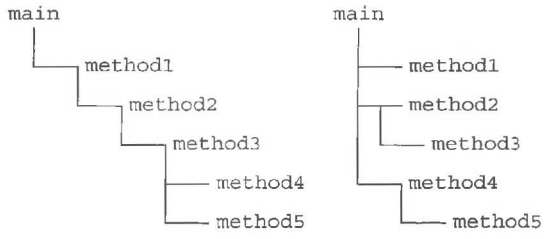


Figure 4.8 Sample code with chaining (left) and without chaining (right)

The `BadBMI` program suffers heavily from chaining. Each method does a small amount of work and then calls the next method, passing more and more parameters down the chain. The `main` method calls `person`, which calls `getWeight`, which calls `reportStatus`. Never does the flow of execution return to `main` in the middle of the computation. So when you read `main`, you don't get a very clear idea of what computations will be made.

One method should not call another simply as a way of moving on to the next task. A more desirable flow of control is to let `main` manage the overall execution of tasks in the program, as shown in the `BMI3` program and on the right side of Figure 4.8. This guideline doesn't mean that it is always bad for one method to call another method; it is okay for one method to call another when the second is a subtask within the overall task of the first, such as in `BMI3` when the `reportResults` method calls `reportStatus`.

5. Data should be "owned" at the lowest level possible. Decisions in a company should be made at the lowest possible level in the organizational hierarchy. For example, a low-level administrator can decide how to perform his or her own work without needing to constantly consult a manager for approval. But the administrator does not have enough information or expertise to design the entire product line; this design task goes to a higher authority such as the manager. The key principle is that each work task should be given to the lowest person in the hierarchy who can correctly handle it.

This principle has two applications in computer programs. The first is that the `main` method should avoid performing low-level tasks as much as possible. For example, in an interactive program `main` should not read the majority of the user input or contain lots of `println` statements.

The second application is that variables should be declared and initialized in the narrowest possible scope. A poor design is for `main` (or another high-level method) to read all of the input, perform heavy computations, and then pass the resulting data as parameters to the various low-level methods. A better design uses low-level methods to read and process the data, and return data to `main` only if they are needed by a later subtask in the program.

It is a sign of poor data ownership when the same parameter must be passed down several method calls, such as the `height` variable in the `BadBMI` program. If you are passing the same parameter down several levels of calls, perhaps that piece of data should instead be read and initialized by one of the lower-level methods (unless it is a shared object such as a `Scanner`).

Chapter Summary

An `if` statement lets you write code that will execute only if a certain condition is met. An `if/else` statement lets you execute one piece of code if a condition is met, and another if the condition is not met. Conditions are Boolean expressions and can be written using relational operators such as `<`, `>=`, and `!=`. You can test multiple conditions using the `&&` and `||` operators.

You can nest `if/else` statements to test a series of conditions and execute the appropriate block of code on the basis of whichever condition is true.

The `==` operator that tests primitive data for equality doesn't behave the way we would expect with objects, so we test objects for equality by calling their `equals` method instead.

Common code that appears in every branch of an `if/else` statement should be factored out so that it is not replicated multiple times in the code.

Cumulative algorithms compute values incrementally. A cumulative sum loop declares a sum variable and incrementally adds to that variable's value inside the loop.

Since the `double` type does not store all values exactly, small roundoff errors can occur when the computer performs calculations on real numbers. Avoid these errors by providing a small amount of tolerance in your code for values near the values that you expect.

The `char` type represents individual characters of text. Each letter of a `String` is stored internally as a `char` value, and you can use the `String`'s `charAt` method to access these characters with an index.

The `System.out.printf` method prints formatted text. You can specify complex format strings to control the width, alignment, and precision by which values are printed.

You can "throw" (generate) exceptions in your own code. This technique can be useful if your code ever reaches an unrecoverable error condition, such as the passing of an invalid argument value to a method.

Self-Check Problems

Section 4.1: `if/else` Statements

1. Translate each of the following English statements into logical tests that could be used in an `if/else` statement. Write the appropriate `if` statement with your logical test. Assume that three `int` variables, `x`, `y`, and `z`, have been declared.

   a. `z` is odd.
   b. `z` is not greater than `y`'s square root.
   c. `y` is positive.
   d. Either `x` or `y` is even, and the other is odd.
   e. `y` is a multiple of `z`.
   f. `z` is not zero.
   g. `y` is greater in magnitude than `z`.
   h. `x` and `z` are of opposite signs.
   i. `y` is a nonnegative one-digit number.

j. z is nonnegative.

k. x is even.

l. x is closer in value to y than z is.

2. Given the variable declarations

```
int x = 4;
int y = -3;
int z = 4;
```

what are the results of the following relational expressions?

a. `x == 4`

b. `x == y`

c. `x == z`

d. `y == z`

e. `x + y > 0`

f. `x - z != 0`

g. `y * y <= z`

h. `y / y == 1`

i. `x * (y + 2) > y - (y + z) * 2`

3. Which of the following `if` statement headers uses the correct syntax?

a. `if x = 10 then {`

b. `if [x == 10] {`

c. `if (x => y) {`

d. `if (x equals 42) {`

e. `if (x == y) {`

4. The following program contains 7 mistakes! What are they?

```
1  public class Oops4 {
2      public static void main(String[] args) {
3          int a = 7, b = 42;
4          minimum(a, b);
5          if (smaller = a) {
6              System.out.println("a is the smallest!");
7          }
8      }
9
10     public static void minimum(int a, int b) {
11         if (a < b) {
12             int smaller = a;
13         } else (a => b) {
14             int smaller = b;
15         }
16         return int smaller;
17     }
18 }
```

5. Consider the following method:

```
public static void ifElseMystery1(int x, int y) {
    int z = 4;
    if (z <= x) {
        z = x + 1;
    } else {
        z = z + 9;
    }
    if (z <= y) {
        y++;
    }
    System.out.println(z + " " + y);
}
```

What output is produced for each of the following calls?

a. `ifElseMystery1(3, 20);`
b. `ifElseMystery1(4, 5);`
c. `ifElseMystery1(5, 5);`
d. `ifElseMystery1(6, 10);`

6. Consider the following method:

```
public static void ifElseMystery2(int a, int b) {
    if (a * 2 < b) {
        a = a * 3;
    } else if (a > b) {
        b = b + 3;
    }
    if (b < a) {
        b++;
    } else {
        a--;
    }
    System.out.println(a + " " + b);
}
```

What output is produced for each of the following calls?

a. `ifElseMystery2(10, 2);`
b. `ifElseMystery2(3, 8);`
c. `ifElseMystery2(4, 4);`
d. `ifElseMystery2(10, 30);`

7. Write Java code to read an integer from the user, then print even if that number is an even number or odd otherwise. You may assume that the user types a valid integer.

8. The following code contains a logic error:

```
Scanner console = new Scanner(System.in);
System.out.print("Type a number: ");
int number = console.nextInt();
if (number % 2 == 0) {
    if (number % 3 == 0) {
        System.out.println("Divisible by 6.");
    } else {
        System.out.println("Odd.");
    }
}
```

Examine the code and describe a case in which the code would print something that is untrue about the number that was entered. Explain why. Then correct the logic error in the code.

9. Describe a problem with the following code:

```
Scanner console = new Scanner(System.in);
System.out.print("What is your favorite color?");
String name = console.next();
if (name == "blue") {
    System.out.println("Mine, too!");
}
```

10. Factor out redundant code from the following example by moving it out of the `if/else` statement, preserving the same output.

```
if (x < 30) {
    a = 2;
    x++;
    System.out.println("Java is awesome! " + x);
} else {
    a = 2;
    System.out.println("Java is awesome! " + x);
}
```

11. The following code is poorly structured:

```
int sum = 1000;
Scanner console = new Scanner(System.in);
System.out.print("Is your money multiplied 1 or 2 times? ");
int times = console.nextInt();
if (times == 1) {
    System.out.print("And how much are you contributing? ");
    int donation = console.nextInt();
    sum = sum + donation;
    count1++;
    total = total + donation;
}
```

```
if (times == 2) {
    System.out.print("And how much are you contributing? ");
    int donation = console.nextInt();
    sum = sum + 2 * donation;
    count2++;
    total = total + donation;
}
```

Rewrite it so that it has a better structure and avoids redundancy. To simplify things, you may assume that the user always types 1 or 2. (How would the code need to be modified to handle any number that the user might type?)

12. The following code is poorly structured:

```
Scanner console = new Scanner(System.in);
System.out.print("How much will John be spending? ");
double amount = console.nextDouble();
System.out.println();
int numBills1 = (int) (amount / 20.0);
if (numBills1 * 20.0 < amount) {
    numBills1++;
}
System.out.print("How much will Jane be spending? ");
amount = console.nextDouble();
System.out.println();
int numBills2 = (int) (amount / 20.0);
if (numBills2 * 20.0 < amount) {
    numBills2++;
}
System.out.println("John needs " + numBills1 + " bills");
System.out.println("Jane needs " + numBills2 + " bills");
```

Rewrite it so that it has a better structure and avoids redundancy. You may wish to introduce a method to help capture redundant code.

13. Write a piece of code that reads a shorthand text description of a color and prints the longer equivalent. Acceptable color names are `B` for `Blue`, `G` for `Green`, and `R` for `Red`. If the user types something other than `B`, `G`, or `R`, the program should print an error message. Make your program case-insensitive so that the user can type an uppercase or lowercase letter. Here are some example executions:

```
What color do you want? B
You have chosen Blue.

What color do you want? g
You have chosen Green.

What color do you want? Bork
Unknown color: Bork
```

14. Write a piece of code that reads a shorthand text description of a playing card and prints the longhand equivalent. The shorthand description is the card's rank (`2` through `10`, `J`, `Q`, `K`, or `A`) followed by its suit (`C`, `D`, `H`, or `S`). You should expand the shorthand into the form "<Rank> of <Suit>". You may assume that the user types valid input. Here are two sample executions:

```
Enter a card: 9 S
Nine of Spades

Enter a card: K C
King of Clubs
```

Section 4.2: Cumulative Algorithms

15. What is wrong with the following code, which attempts to add all numbers from 1 to a given maximum? Describe how to fix the code.

```
public static int sumTo(int n) {
    for (int i = 1; i <= n; i++) {
        int sum = 0;
        sum += i;
    }
    return sum;
}
```

16. What is wrong with the following code, which attempts to return the number of factors of a given integer *n*? Describe how to fix the code.

```
public static int countFactors(int n) {
    for (int i = 1; i <= n; i++) {
        if (n % i == 0) {  // factor
            return i;
        }
    }
}
```

17. Write code to produce a cumulative product by multiplying together many numbers that are read from the console.

18. The following expression should equal `6.8`, but in Java it does not. Why not?

```
0.2 + 1.2 + 2.2 + 3.2
```

19. The following code was intended to print a message, but it actually produces no output. Describe how to fix the code to print the expected message.

```
double gpa = 3.2;
if (gpa * 3 == 9.6) {
   System.out.println("You earned enough credits.");
}
```

Section 4.3: Text Processing

20. What output is produced by the following program?

```
1 public class CharMystery {
2     public static void printRange(char startLetter, char endLetter) {
3         for (char letter = startLetter; letter <= endLetter; letter++) {
4             System.out.print(letter);
5         }
6         System.out.println();
7     }
8
9     public static void main(String[] args) {
10        printRange('e', 'g');
11        printRange('n', 's');
12        printRange('z', 'a');
13        printRange('q', 'r');
14    }
15 }
```

21. Write an `if` statement that tests to see whether a `String` begins with a capital letter.

22. What is wrong with the following code, which attempts to count the number occurrences of the letter `'e'` in a `String`, case-insensitively?

```
int count = 0;
for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i).toLowerCase() == 'e') {
        count++;
    }
}
```

23. Consider a `String` stored in a variable called `name` that stores a person's first and last name (e.g., "Marla Singer"). Write the expression that would produce the last name followed by the first initial (e.g., "Singer, M.").

24. Write code to examine a `String` and determine how many of its letters come from the second half of the alphabet (that is, have values of `'n'` or subsequent letters). Compare case-insensitively, such that values of `'N'` through `'Z'` also count. Assume that every character in the `String` is a letter.

Section 4.4: Methods with Conditional Execution

25. Consider a method `printTriangleType` that accepts three integer arguments representing the lengths of the sides of a triangle and prints the type of triangle that these sides form. The three types are equilateral, isosceles, and scalene. An equilateral triangle has three sides of the same length, an isosceles triangle has two sides that are the same length, and a scalene triangle has three sides of different lengths.

However, certain integer values (or combinations of values) would be illegal and could not represent the sides of an actual triangle. What are these values? How would you describe the precondition(s) of the `printTriangleType` method?

26. Consider a method `getGrade` that accepts an integer representing a student's grade percentage in a course and returns that student's numerical course grade. The grade can be between `0.0` (failing) and `4.0` (perfect). What are the preconditions of such a method?

27. The following method attempts to return the median (middle) of three integer values, but it contains logic errors. In what cases does the method return an incorrect result? How can the code be fixed?

```
public static int medianOf3(int n1, int n2, int n3) {
    if (n1 < n2) {
        if (n2 < n3) {
            return n2;
        } else {
            return n3;
        }
    } else {
        if (n1 < n3) {
            return n1;
        } else {
            return n3;
        }
    }
}
```

28. One of the exercises in Chapter 3 asked you to write a method that would find the roots of a quadratic equation of the form $ax^2 + bx + c = 0$. The quadratic method was passed `a`, `b`, and `c` and then applied the following quadratic formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Under what conditions would this formula fail? Modify the quadratic method so that it will reject invalid values of `a`, `b`, or `c` by throwing an exception. (If you did not complete the exercise in the previous chapter, just write the method's header and the exception-throwing code.)

29. Consider the following Java method, which is written incorrectly:

```
// This method should return how many of its three
// arguments are odd numbers.
public static void printNumOdd(int n1, int n2, int n3) {
   int count = 0;
   if (n1 % 2 != 0) {
      count++;
   } else if (n2 % 2 != 0) {
      count++;
```

```
    } else if (n3 % 2 != 0) {
        count++;
    }
    System.out.println(count + " of the 3 numbers are odd.");
}
```

Under what cases will the method print the correct answer, and when will it print an incorrect answer? What should be changed to fix the code? Can you think of a way to write the code correctly without any `if/else` statements?

Exercises

1. Write a method called `fractionSum` that accepts an integer parameter n and returns as a `double` the sum of the first n terms of the sequence

$$\sum_{i=1}^{n} \frac{1}{i}$$

In other words, the method should generate the following sequence:

$$1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \frac{1}{5} + \cdots$$

You may assume that the parameter n is nonnegative.

2. Write a method called `repl` that accepts a `String` and a number of repetitions as parameters and returns the `String` concatenated that many times. For example, the call `repl("hello", 3)` should return `"hellohellohello"`. If the number of repetitions is zero or less, the method should return an empty string.

3. Write a method called `season` that takes as parameters two integers representing a month and day and returns a `String` indicating the season for that month and day. Assume that the month is specified as an integer between 1 and 12 (1 for January, 2 for February, and so on) and that the day of the month is a number between 1 and 31. If the date falls between 12/16 and 3/15, the method should return `"winter"`. If the date falls between 3/16 and 6/15, the method should return `"spring"`. If the date falls between 6/16 and 9/15, the method should return `"summer"`. And if the date falls between 9/16 and 12/15, the method should return `"fall"`.

4. Write a method called `daysInMonth` that takes a month (an integer between 1 and 12) as a parameter and returns the number of days in that month in this year. For example, the call `daysInMonth(9)` would return `30` because September has 30 days. Assume that the code is not being run during a leap year (that February always has 28 days). The following table lists the number of days in each month:

Month	1 Jan	2 Feb	3 Mar	4 Apr	5 May	6 Jun	7 Jul	8 Aug	9 Sep	10 Oct	11 Nov	12 Dec
Days	31	28	31	30	31	30	31	31	30	31	30	31

5. Write a method called `pow` that accepts a base and an exponent as parameters and returns the base raised to the given power. For example, the call `pow(3, 4)` should return `3 * 3 * 3 * 3`, or `81`. Assume that the base and exponent are nonnegative.

6. Write a method called `printRange` that accepts two integers as arguments and prints the sequence of numbers between the two arguments, separated by spaces. Print an increasing sequence if the first argument is smaller than the second; otherwise, print a decreasing sequence. If the two numbers are the same, that number should be printed by itself. Here are some sample calls to `printRange`:

```
printRange(2, 7);
printRange(19, 11);
printRange(5, 5);
```

The output produced from these calls should be the following sequences of numbers:

```
2 3 4 5 6 7
19 18 17 16 15 14 13 12 11
5
```

7. Write a static method called `xo` that accepts an integer *size* as a parameter and prints a square of *size* by *size* characters, where all characters are "o" except that an "x" pattern of "x" characters has been drawn from the corners of the square. On the first line, the first and last characters are "x"; on the second line, the second and second-from-last characters are "x"; and so on. Here are two example outputs:

`xo(5);`	`xo(6);`
`xooox`	`xoooox`
`oxoxo`	`oxooxo`
`ooxoo`	`ooxxoo`
`oxoxo`	`ooxxoo`
`xooox`	`oxooxo`
	`xoooox`

8. Write a method called `smallestLargest` that accepts a `Scanner` for the console as a parameter and asks the user to enter numbers, then prints the smallest and largest of all the numbers supplied by the user. You may assume that the user enters a valid number greater than `0` for the number of numbers to read. Here is a sample execution:

```
How many numbers do you want to enter? 4
Number 1: 5
Number 2: 11
Number 3: -2
Number 4: 3
Smallest = -2
Largest = 11
```

9. Write a method called `evenSumMax` that accepts a `Scanner` for the console as a parameter. The method should prompt the user for a number of integers, then prompt the integer that many times. Once the user has entered all the integers, the method should print the sum of all the even numbers the user typed, along with the largest even number typed. You may assume that the user will type at least one nonnegative even integer. Here is an example dialogue:

```
How many integers? 4
Next integer? 2
```

```
Next integer? 9
Next integer? 18
Next integer? 4
Even sum = 24, Even max = 18
```

10. Write a method called `printGPA` that accepts a `Scanner` for the console as a parameter and calculates a student's grade point average. The user will type a line of input containing the student's name, then a number that represents the number of scores, followed by that many integer scores. Here are two example dialogues:

```
Enter a student record: Maria 5 72 91 84 89 78
Maria's grade is 82.8

Enter a student record: Jordan 4 86 71 62 90
Jordan's grade is 77.25
```

Maria's grade is 82.8 because her average of (72 + 91 + 84 + 89 + 78) / 5 equals 82.8.

11. Write a method called `longestName` that accepts a `Scanner` for the console and an integer *n* as parameters and prompts for *n* names, then prints the longest name (the name that contains the most characters) in the format shown below, which might result from a call of `longestName(console, 4)`:

```
name #1? Roy
name #2? DANE
name #3? sTeFaNiE
name #4? Mariana
Stefanie's name is longest
```

12. Write the method called `printTriangleType` referred to in Self-Check Problem 25. This method accepts three integer arguments representing the lengths of the sides of a triangle and prints the type of triangle that these sides form. Here are some sample calls to `printTriangleType`:

```
printTriangleType(5, 7, 7);
printTriangleType(6, 6, 6);
printTriangleType(5, 7, 8);
printTriangleType(2, 18, 2);
```

The output produced by these calls should be

```
isosceles
equilateral
scalene
isosceles
```

Your method should throw an `IllegalArgumentException` if passed invalid values, such as ones where one side's length is longer than the sum of the other two, which is impossible in a triangle. For example, the call of `printTriangleType(2, 18, 2);` should throw an exception.

13. Write a method called `average` that takes two integers as parameters and returns the average of the two integers.

14. Modify your pow method from Exercise 5 to make a new method called pow2 that uses the type `double` for the first parameter and that works correctly for negative numbers. For example, the call `pow2(-4.0, 3)` should return `-4.0 * -4.0 * -4.0`, or `-64.0`, and the call `pow2(4.0, -2)` should return `1 / 16`, or `0.0625`.

15. Write a method called `getGrade` that accepts an integer representing a student's grade in a course and returns that student's numerical course grade. The grade can be between `0.0` (failing) and `4.0` (perfect). Assume that scores are in the range of `0` to `100` and that grades are based on the following scale:

Score	Grade
< 60	0.0
60–62	0.7
63	0.8
64	0.9
65	1.0
. . .	
92	3.7
93	3.8
94	3.9
>= 95	4.0

For an added challenge, make your method throw an `IllegalArgumentException` if the user passes a grade lower than `0` or higher than `100`.

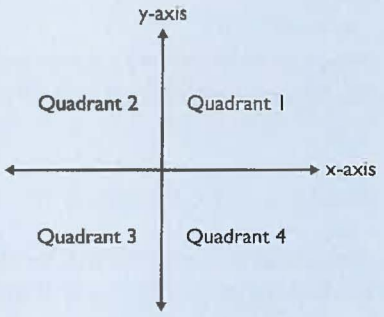
16. Write a method called `printPalindrome` that accepts a `Scanner` for the console as a parameter, prompts the user to enter one or more words, and prints whether the entered `String` is a palindrome (i.e., reads the same forward as it does backward, like `"abba"` or `"racecar"`).

For an added challenge, make the code case-insensitive, so that words like "Abba" and "Madam" will be considered palindromes.

17. Write a method called `swapPairs` that accepts a `String` as a parameter and returns that `String` with each pair of adjacent letters reversed. If the `String` has an odd number of letters, the last letter is unchanged. For example, the call `swapPairs("example")` should return `"xemalpe"` and the call `swapPairs("hello there")` should return `"ehll ohtree"`.

18. Write a method called `wordCount` that accepts a `String` as its parameter and returns the number of words in the `String`. A word is a sequence of one or more nonspace characters (any character other than `' '`). For example, the call `wordCount("hello")` should return `1`, the call `wordCount("how are you?")` should return `3`, the call `wordCount(" this string has wide spaces ")` should return `5`, and the call `wordCount(" ")` should return `0`.

19. Write a method called `quadrant` that accepts as parameters a pair of `double` values representing an (x, y) point and returns the quadrant number for that point. Recall that quadrants are numbered as integers from 1 to 4 with the upper-right quadrant numbered 1 and the subsequent quadrants numbered in a counterclockwise fashion:



Notice that the quadrant is determined by whether the x and y coordinates are positive or negative numbers. Return `0` if the point lies on the x- or y-axis. For example, the call of `quadrant(-2.3, 3.5)` should return `2` and the call of `quadrant(4.5, -4.5)` should return `4`.

20. Write a method called `numUnique` that takes three integers as parameters and returns the number of unique integers among the three. For example, the call `numUnique(18, 3, 4)` should return `3` because the parameters have three different values. By contrast, the call `numUnique(6, 7, 6)` should return `2` because there are only two unique numbers among the three parameters: `6` and `7`.

21. Write a method called `perfectNumbers` that accepts an integer maximum as its parameter and prints all "perfect numbers" up to and including that maximum. A perfect number is an integer that is equal to the sum of its proper factors, that is, all numbers that evenly divide it other than 1 and itself. For example, 28 is a perfect number because $1 + 2 + 4 + 7 + 14 = 28$. The call `perfectNumbers(500);` should produce the following output:

```
Perfect numbers up to 500: 6 28 496
```

Programming Projects

1. Write a program that prompts for a number and displays it in Roman numerals.

2. Write a program that prompts for a date (month, day, year) and reports the day of the week for that date. It might be helpful to know that January 1, 1601, was a Monday.

3. Write a program that compares two college applicants. The program should prompt for each student's GPA, SAT, and ACT exam scores and report which candidate is more qualified on the basis of these scores.

4. Write a program that prompts for two people's birthdays (month and day), along with today's month and day. The program should figure out how many days remain until each user's birthday and which birthday is sooner. Hint: It is much easier to solve this problem if you convert each date into an "absolute day" of year, from 1 through 365.

5. Write a program that computes a student's grade in a course. The course grade has three components: homework assignments, a midterm exam, and a final exam. The program should prompt the user for all information necessary to compute the grade, such as the number of homework assignments, the points earned and points possible for each assignment, the midterm and final exam scores, and whether each exam was curved (and, if so, by how much).

   Consider writing a variation of this program that reports what final exam score the student needs to get a certain course grade.

6. A useful technique for catching typing errors is to use a check digit. For example, suppose that a school assigns a six-digit number to each student. A seventh digit can be determined from the other digits with the use of the following formula:

   7th digit = (1 * (1st digit) + 2 * (2nd digit) + . . . + 6 * (6th digit)) % 10

   When a user types in a student number, the user types all seven digits. If the number is typed incorrectly, the check digit will fail to match in 90% of the cases. Write an interactive program that prompts for a six-digit student number and reports the check digit for that number, using the preceding formula.

7. Write a program that displays Pascal's triangle:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5  10  10   5   1
    1   6  15  20  15   6   1
  1   7  21  35  35  21   7   1
1   8  28  56  70  56  28   8   1
  1   9  36   84 126 126  84  36   9  1
1  10  45 120 210 252 210 120  45  10   1
```

   Use `System.out.printf` to format the output into fields of width 4.