

Program Logic and Indefinite Loops

Introduction

The chapter begins by examining a new construct called a `while` loop that allows you to loop an indefinite number of times. The `while` loop will allow you to solve a new class of programming problems in which you don't know in advance how many times you want a loop to execute. For example, game-playing programs often involve `while` loops because it is not possible to know beforehand how the user will play the game. Because we will be exploring game programs, we will also explore how to generate random numbers inside a Java program. We will also explore another class of algorithms known as fencepost algorithms that occur often in loop-programming tasks.

The chapter then discusses the fourth primitive type that we are going to examine in detail, `boolean`. The `boolean` type is used to store logical (true/false) information. Once you understand the details of the `boolean` type, you will be able to write complex loops involving multiple tests.

Next, we'll briefly examine the important topic of handling user errors.

The chapter concludes with a discussion of assertions. Using assertions, you can reason about the formal properties of programs (what is true at different points in program execution).

5.1 The `while` Loop

- A Loop to Find the Smallest Divisor
- Random Numbers
- Simulations
- The `do/while` Loop

5.2 Fencepost Algorithms

- Sentinel Loops
- Fencepost with `if`

5.3 The `boolean` Type

- Logical Operators
- Short-Circuited Evaluation
- `boolean` Variables and Flags
- Boolean Zen
- Negating Boolean Expressions

5.4 User Errors

- Scanner Lookahead
- Handling User Errors

5.5 Assertions and Program Logic

- Reasoning about Assertions
- A Detailed Assertions Example

5.6 Case Study: `NumberGuess`

- Initial Version without Hinting
- Randomized Version with Hinting
- Final Robust Version

5.1 The while Loop

The for loops we have been writing since Chapter 2 are fairly simple loops that execute a predictable number of times. Recall that we call them *definite* loops because we know before the loops begin executing exactly how many times they will execute. Now we want to turn our attention to *indefinite* loops, which execute an unknown number of times. Indefinite loops come up often in interactive programs and file processing. For example, you don't know in advance how many times a user might want to play a game, and you won't know before you look at a file exactly how much data it stores.

The while loop is the first indefinite loop we will study. It has the following syntax:

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

The diagram in Figure 5.1 indicates the flow of control for the while loop. The loop performs its test and, if the test evaluates to *true*, executes the controlled statements. It repeatedly tests and executes if the test evaluates to *true*. Only when the test evaluates to *false* does the loop terminate.

As Figure 5.1 indicates, the while loop performs its test at the top of the loop, before the body of the loop is executed. A while loop will not execute its controlled statements if its test evaluates to *false* the first time it is evaluated.

Here is an example of a while loop:

```
int number = 1;  
while (number <= 200) {  
    number = number * 2;  
}
```

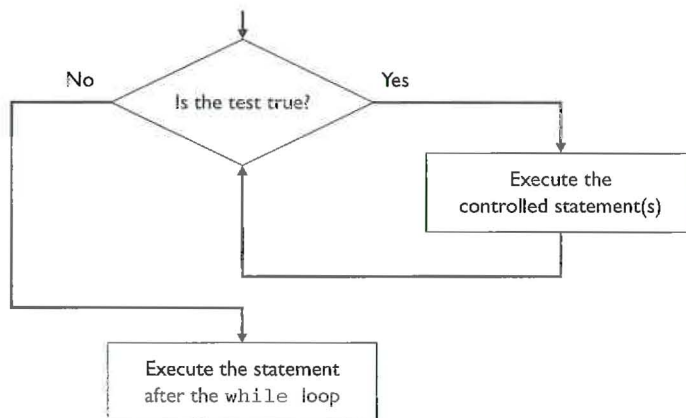


Figure 5.1 Flow of while loop

This loop initializes an integer variable called `number` to 1 and then doubles it while it is less than or equal to 200. On the surface, this operation is similar to using an `if` statement:

```
int number = 1;
if (number <= 200) {
    number = number * 2;
}
```

The difference between the two forms is that the `while` loop executes multiple times, looping until the test evaluates to `false`. The `if` statement executes the doubling statement only once, leaving `number` equal to 2. The `while` loop executes the doubling statement repeatedly, with `number` taking on the values 1, 2, 4, 8, 16, 32, 64, 128, and 256. The loop doesn't stop executing until the test evaluates to `false`. It executes the assignment statement eight times and terminates when `number` is set to the value 256 (the first power of 2 that is greater than 200).

Here is a `while` loop containing two statements:

```
int number = 1;
while (number <= max) {
    System.out.println("Hi there");
    number++;
}
```

This `while` loop is almost the same as the following `for` loop:

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

The only difference between these two loops is the scope of the variable `number`. In the `while` loop, `number` is declared in the scope outside the loop. In the `for` loop, `number` is declared inside the loop.

A Loop to Find the Smallest Divisor

Suppose you want to find the smallest divisor of a number other than 1. Table 5.1 gives examples of what you are looking for.

Here is a pseudocode description of how you might find this value:

```
start divisor at 2.
while (the current value of divisor does not work) {
    increase divisor.
}
```

Table 5.1 Examples of Factors

Number	Factors	Smallest divisor
10	2 * 5	2
15	3 * 5	3
25	5 * 5	5
31	31	31
77	7 * 11	7

You don't start `divisor` at 1 because you are looking for the first divisor greater than 1. To refine this pseudocode, you must be more explicit about what makes a divisor work. A divisor of a number has no remainder when the number is divided by that divisor. You can rewrite this rule as the following pseudocode:

```
start divisor at 2.
while (the remainder of number/divisor is not 0) {
    increase divisor.
}
```

This exercise is a use for the mod operator, which gives the remainder for integer division. The following while loop performs the task:

```
int divisor = 2;
while (number % divisor != 0) {
    divisor++;
}
```

One problem you will undoubtedly encounter when you write while loops is the infamous infinite loop. Consider the following code:

```
 int number = 1;
while (number > 0) {
    number++;
}
```

Because `number` begins as a positive value and the loop makes it larger, this loop will continue indefinitely. You must be careful when you formulate your while loops to avoid situations in which a piece of code will never finish executing. Every time you write a while loop, you should consider when and how it will finish executing.

Common Programming Error**Infinite Loop**

It is relatively easy to write a while loop that never terminates. One reason it's so easy to make this mistake is that a while loop doesn't have an update step in its header like a for loop does. It's crucial for the programmer to include a correct update step because this step is needed to eventually cause the loop's test to fail.

Consider the following code, which is intended to prompt the user for a number and repeatedly print that number divided in half until 0 is reached. This first attempt doesn't compile:

```
Scanner console = new Scanner(System.in);
System.out.print("Type a number: ");

// this code does not compile
while (number > 0) {
    int number = console.nextInt();
    System.out.println(number / 2);
}
```

The problem with the preceding code is that the variable `number` needs to be in scope during the loop's test, so it cannot be declared inside the loop. An incorrect attempt to fix this compiler error would be to cut and paste the line initializing `number` outside the loop:

```
// this code has an infinite loop
int number = console.nextInt(); // moved out of loop

while (number > 0) {
    System.out.println(number / 2);
}
```

This version of the code has an infinite loop; if the loop is entered, it will never be exited. This problem arises because there is **no update** inside the while loop's body to change the value of `number`. If `number` is greater than 0, the loop will keep printing its value and checking the loop test, and the test will evaluate to true every time.

Continued on next page

Continued from previous page

The following version of the code solves the infinite loop problem. The loop contains an update step on each pass that divides the integer in half and stores its new value. If the integer hasn't reached 0, the loop repeats:

```
// this code behaves correctly
int number = console.nextInt(); // moved out of loop
while (number > 0) {
    number = number / 2; // update step: divide in half
    System.out.println(number);
}
```

The key idea is that every `while` loop's body should contain code to update the terms that are tested in the loop test. If the `while` loop test examines a variable's value, the loop body should potentially reassign a meaningful new value to that variable.

Random Numbers

We often want our programs to exhibit apparently random behavior. For example, we want game-playing programs to make up a number for the user to guess, shuffle a deck of cards, pick a word from a list for the user to guess, and so on. Programs are, by their very nature, predictable and nonrandom. But we can produce values that seem to be random. Such values are called *pseudorandom* because they are produced algorithmically.

Pseudorandom Numbers

Numbers that, although they are derived from predictable and well-defined algorithms, mimic the properties of numbers chosen at random.

Java provides several mechanisms for obtaining pseudorandom numbers. One option is to call the `random` method from the `Math` class to obtain a random value of type `double` that has the following property:

$$0.0 \leq \text{Math.random}() < 1.0$$

This method provides a quick and easy way to get a random number, and you can use multiplication to change the range of the numbers the method produces. Java also provides a class called `Random` that can be easier to use. It is included in the `java.util` package, so you have to include an import declaration at the beginning of your program to use it.

`Random` objects have several useful methods that are related to generating pseudorandom numbers, listed in Table 5.2. Each time you call one of these methods, Java will generate and return a new random number of the requested type.

Table 5.2 Useful Methods of Random Objects

Method	Description
<code>nextInt()</code>	Random integer between -2^{31} and $(2^{31} - 1)$
<code>nextInt(max)</code>	Random integer between 0 and $(\text{max} - 1)$
<code>nextDouble()</code>	Random real number between 0.0 (inclusive) and 1.0 (exclusive)
<code>nextBoolean()</code>	Random logical value of <code>true</code> or <code>false</code>

To create random numbers, you first construct a `Random` object:

```
Random r = new Random();
```

You can then call its `nextInt` method, passing it a maximum integer. The number returned will be between 0 (inclusive) and the maximum (exclusive). For example, if you call `nextInt(100)`, you will get a number between 0 and 99. You can add 1 to the number to have a range between 1 and 100.

Let's look at a simple program that picks numbers between 1 and 10 until a particular number comes up. We'll use the `Random` class to construct an object for generating our pseudorandom numbers.

Our loop should look something like this (where `number` is the value the user has asked us to generate):

```
int result;
while (result != number) {
    result = r.nextInt(10) + 1; // random number from 1–10
    System.out.println("next number = " + result);
}
```

Notice that we have to declare the variable `result` outside the `while` loop, because `result` appears in the `while` loop test. The preceding code has the right approach, but Java won't accept it. The code generates an error message that the variable `result` might not be initialized. This is an example of a loop that needs *priming*.

Priming a Loop

Initializing variables before a loop to “prime the pump” and guarantee that the loop is entered.

We want to set the variable `result` to something that will cause the loop to be entered, but the value isn't important as long as it gets us into the loop. We do want to be careful not to set it to a value the user wants us to generate, though. We are dealing with values between 1 and 10 in this program, so we could set `result` to a value such as -1 that is clearly outside this range of numbers. We sometimes refer to this as

a “dummy” value because we don’t actually process it. Later in this chapter we will see a variation of the while loop that doesn’t require this kind of priming.

The following is the complete program solution:

```

1 import java.util.*;
2
3 public class Pick {
4     public static void main(String[] args) {
5         System.out.println("This program picks numbers from");
6         System.out.println("1 to 10 until a particular");
7         System.out.println("number comes up.");
8         System.out.println();
9
10        Scanner console = new Scanner(System.in);
11        Random r = new Random();
12
13        System.out.print("Pick a number between 1 and 10--> ");
14        int number = console.nextInt();
15
16        int result = -1; // set to -1 to make sure we enter the loop
17        int count = 0;
18        while (result != number) {
19            result = r.nextInt(10) + 1; // random number from 1-10
20            System.out.println("next number = " + result);
21            count++;
22        }
23        System.out.println("Your number came up after " +
24                            count + " times");
25    }
26 }

```

Depending on the sequence of numbers returned by the Random object, the program might end up picking the given number quickly, as in the following sample execution:

```

This program picks numbers from
1 to 10 until a particular
number comes up.

Pick a number between 1 and 10--> 2
next number = 7
next number = 8
next number = 2
Your number came up after 3 times

```

It's also possible that the program will take a while to pick the number, as in the following sample execution:

```
This program picks numbers from
1 to 10 until a particular
number comes up.

Pick a number between 1 and 10--> 10
next number = 9
next number = 7
next number = 7
next number = 5
next number = 8
next number = 8
next number = 1
next number = 5
next number = 1
next number = 9
next number = 7
next number = 10
Your number came up after 12 times
```

Common Programming Error

Misusing the Random Object

A Random object chooses a new random integer every time the nextInt method is called. When students are trying to produce a constrained random value, such as one that is odd, sometimes they mistakenly write code such as the following:

```
// this code contains a bug
Random r = new Random();

if (r.nextInt() % 2 == 0) {
    System.out.println("Even number: " + r.nextInt());
} else {
    System.out.println("Odd number: " + r.nextInt());
}
```

The preceding code fails in many cases because the Random object produces one random integer for use in the if/else test, then another for use in whichever println statement is chosen to execute. For example, the if test might retrieve a random value of 47 from the Random object. The test would find

Continued on next page

Continued from previous page

that $47 \% 2$ does not equal 0, so the code would proceed to the `else` statement. The `println` statement would then execute another call on `nextInt`, which would return a completely different number (say, 128). The output of the code would then be the following bizarre statement:

```
Odd number: 128
```

The solution to this problem is to store the randomly chosen integer in a variable and call `nextInt` again only if another random integer is truly needed. The following code accomplishes this task:

```
// this code behaves correctly
Random r = new Random();
int n = r.nextInt(); // save random number into a variable
if (n % 2 == 0) {
    System.out.println("Even number: " + n);
} else {
    System.out.println("Odd number: " + n);
}
```

Simulations



Traditional science and engineering involve a lot of real-world interaction. Scientists run experiments to test their hypotheses and engineers build prototypes to test their designs. But increasingly scientists and engineers are turning to computers as a way to increase their productivity by running simulations first to explore possibilities before they go out and run an actual experiment or build an actual prototype. A famous computer scientist named Jeanette Wing has argued that this increased use of computation by scientists and engineers will lead to computational thinking being viewed as fundamental in the same way that reading, writing, and arithmetic are considered fundamental today.

From a programming perspective, the two key ingredients in a simulation are pseudorandom numbers and loops. Some simulations can be written using `for` loops, but more often than not we use a `while` loop because the simulation should be run indefinitely until some condition is met.

As a simple example, let's look at how we would simulate the rolling of two dice until the sum of the dice is 7. We can use a `Random` object to simulate the dice, calling it once for each of the two dice. We want to loop until the sum is equal to 7 and we can print the various rolls that come up as we run the simulation. Here is a good first attempt:

```

Random r = new Random();
while (sum != 7) {
    // roll the dice once
    int roll1 = r.nextInt(6) + 1;
    int roll2 = r.nextInt(6) + 1;
    int sum = roll1 + roll2;
    System.out.println(roll1 + " + " + roll2 + " = " + sum);
}

```

The preceding code produces the following compiler error:

```

Dice.java:7: error: cannot find symbol
symbol : variable sum
location: class Dice
    while (sum != 7) {
        ^
1 error

```

The problem is that the `while` loop test refers to the variable `sum`, but the variable is declared inside the body of the loop. We can't declare the variable in the inner scope because we need to refer to it in the loop test. So we have to move the variable declaration before the loop. We also have to give the variable an initial value to guarantee that it enters the loop. This code is another example of a time when we need to prime the loop:

```

Random r = new Random();
int sum = 0; // set to 0 to make sure we enter the loop
while (sum != 7) {
    // roll the dice once
    int roll1 = r.nextInt(6) + 1;
    int roll2 = r.nextInt(6) + 1;
    sum = roll1 + roll2;
    System.out.println(roll1 + " + " + roll2 + " = " + sum);
}

```

This version of the code compiles and works properly. A sample execution follows:

```

1 + 4 = 5
5 + 6 = 11
1 + 3 = 4
4 + 3 = 7

```

do/while Loop

The `while` loop is the standard indefinite loop, but Java provides several alternatives. This section presents the `do/while` loop. Other variations are included in Appendix C.

As we have seen, the `while` loop tests at the “top” of the loop, before it executes its controlled statement. Java has an alternative known as the `do/while` loop that tests at the “bottom” of the loop. The `do/while` loop has the following syntax:

```
do {  
    <statement>;  
    ...  
    <statement>;  
} while (<test>;
```

Here is some sample code using a `do/while` loop:

```
int number = 1;  
do {  
    number *= 2;  
} while (number <= 200);
```

This loop produces the same result as the corresponding `while` loop, doubling the variable `number` until its value reaches 256, which is the first power of 2 greater than 200. But unlike the `while` loop, the `do/while` loop always executes its controlled statements at least once. The diagram in Figure 5.2 shows the flow of control in a `do/while` loop.

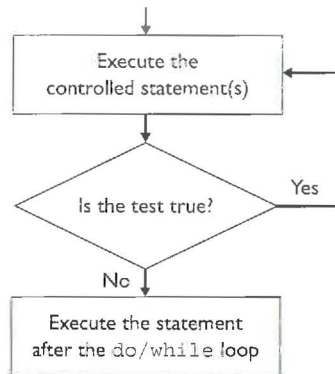


Figure 5.2 Flow of `do/while` loop

The `do/while` loop is most useful in situations in which you know you have to execute the loop at least once. For example, in the last section we wrote the following code that simulates the rolling of two dice until you get a sum of 7:

```
Random r = new Random();  
int sum = 0; // set to 0 to make sure we enter the loop  
while (sum != 7) {  
    // roll the dice once  
    int roll1 = r.nextInt(6) + 1;
```

```

    int roll2 = r.nextInt(6) + 1;
    sum = roll1 + roll2;
    System.out.println(roll1 + " + " + roll2 + " = " + sum);
}

```

We had to prime the loop by setting `sum` to 0 so that the computer would enter the loop. With a `do/while` loop, we can eliminate the priming:

```

Random r = new Random();
int sum;
do {
    // roll the dice once
    int roll1 = r.nextInt(6) + 1;
    int roll2 = r.nextInt(6) + 1;
    sum = roll1 + roll2;
    System.out.println(roll1 + " + " + roll2 + " = " + sum);
} while (sum != 7);

```

In this version, we always execute the body of the loop at least once, which ends up giving a value to the variable `sum` *before* it reaches the loop test that now appears at the bottom of the loop.

There are many programming problems in which using `do/while` loops is appropriate. These loops are often useful in interactive programs where you know you want to do something at least once. For example, you might have a loop that allows a user to play a game multiple times; you can be fairly sure that the user will want to play at least once. Likewise, if you are playing a guessing game with the user, you will always have to obtain at least one guess.

5.2 Fencepost Algorithms



A common programming problem involves a particular kind of loop known as a *fencepost loop*. Consider the following problem: You want to put up a fence that is 100 yards long, and you want to install a post every 10 yards. How many posts do you need? If you do a quick division in your head, you might think that you need 10 posts, but actually you need 11 posts. That's because fences begin and end with posts. In other words, a fence looks like Figure 5.3.

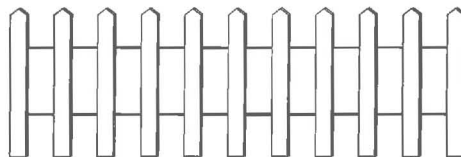


Figure 5.3 A typical fence

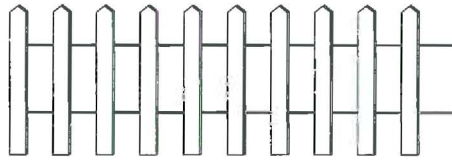


Figure 5.4 A flawed fence

Because you want posts on both the far left and the far right, you can't use the following simple loop (it doesn't plant the final post):

```

for (the length of the fence) {
    plant a post.
    attach some wire.
}
  
```

If you use the preceding loop, you'll get a fence that looks like Figure 5.4.

Switching the order of the two operations doesn't help, because then you miss the first post. The problem with this loop is that it produces the same number of posts as sections of wire, but we know we need an extra post. That's why this problem is also sometimes referred to as the "loop and a half" problem—we want to execute one half of this loop (planting a post) one additional time.

One solution is to plant one of the posts either before or after the loop. The usual solution is to do it before:

```

plant a post.
for (the length of the fence) {
    attach some wire.
    plant a post.
}
  
```

Notice that the order of the two operations in the body of the loop is now reversed because the initial post is planted before the loop is entered.

As a simple example, consider the problem of writing out the integers between 1 and 10, separated by commas. In other words, we want to get the following output:

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

This task is a classic fencepost problem because we want to write out 10 numbers but only 9 commas. In our fencepost terminology, writing a number is the "post" part of the task and writing a comma is the "wire" part. So, implementing the pseudocode we developed, we print the first number before the loop:

```

System.out.print(1);
for (int i = 2; i <= 10; i++) {
  
```



```
        System.out.print(", " + i);  
    }  
    System.out.println();
```

Sentinel Loops



Suppose you want to read a series of numbers from the user and compute their sum. You could ask the user in advance how many numbers to read, as we did in the last chapter, but that isn't always convenient. What if the user has a long list of numbers to enter and hasn't counted them? One way around asking for the number is to pick some special input value that will signal the end of input. We call this a *sentinel value*.

Sentinel

A special value that signals the end of input.

For example, you could tell the user to enter the value -1 to stop entering numbers. But how do you structure your code to use this sentinel? In general, you'll want to use the following approach:

```
sum = 0.  
while (we haven't seen the sentinel) {  
    prompt and read.  
    add it to the sum.  
}
```

This approach doesn't quite work. Suppose, for example, that the user enters the numbers 10, 42, 5, and -1 . As the pseudocode indicates, we'll prompt for and read each of these four values and add them to our sum until we encounter the sentinel value of -1 . We initialize the sum to 0, so this computes $(10 + 42 + 5 + -1)$, which is 56. But the right answer is 57. The sentinel value of -1 isn't supposed to be included in the sum.

This problem is a classic fencepost or "loop-and-a-half" problem: You want to prompt for and read a series of numbers, including the sentinel, and you want to add up most of the numbers, but you don't want to add the sentinel to the sum.

The usual fencepost solution works: We insert the first prompt-and-read instruction before the loop and reverse the order of the two steps in the body of the loop:

```
sum = 0.  
prompt and read.  
while (we haven't seen the sentinel) {  
    add it to the sum.  
    prompt and read.  
}
```

You can then refine this pseudocode by introducing a variable for the number that is read from the user:

```
sum = 0.
prompt and read a value into n.
while (n is not the sentinel) {
    add n to the sum.
    prompt and read a value into n.
}
```

This pseudocode translates fairly easily into Java code:

```
Scanner console = new Scanner(System.in);
int sum = 0;
System.out.print("next integer (-1 to quit)? ");
int number = console.nextInt();
while (number != -1) {
    sum += number;
    System.out.print("next integer (-1 to quit)? ");
    number = console.nextInt();
}
System.out.println("sum = " + sum);
```

When the preceding code is executed, the interaction might look like this:

```
next integer (-1 to quit)? 34
next integer (-1 to quit)? 19
next integer (-1 to quit)? 8
next integer (-1 to quit)? 0
next integer (-1 to quit)? 17
next integer (-1 to quit)? 204
next integer (-1 to quit)? -1
sum = 282
```

Fencepost with **if**

Many of the fencepost loops that you write will require conditional execution. In fact, the fencepost problem itself can be solved with an **if** statement. Remember that the classic solution to the fencepost is to handle the first post before the loop begins:

```
plant a post.
for (the length of the fence) {
    attach some wire.
    plant a post.
}
```

This solution solves the problem, but it can be confusing because inside the loop the steps are apparently in reverse order. You can use an `if` statement and keep the original order of the steps:

```
for (the length of the fence) {  
    plant a post.  
    if (this isn't the last post) {  
        attach some wire.  
    }  
}
```

This variation isn't used as often as the classic solution because it involves both a loop test and a test inside the loop. Often these tests are nearly identical, so it is inefficient to test the same thing twice each time the loop executes. But there will be situations in which you might use this approach. For example, in the classic approach, the lines of code that correspond to planting a post are repeated. If you were writing a program in which this step required a lot of code, you might decide that putting the `if` statement inside the loop was a better approach, even if it led to some extra testing.

As an example, consider writing a method called `multiprint` that will print a string a particular number of times. Suppose that you want the output on a line by itself, inside square brackets, and separated by commas. Here are two example calls:

```
multiprint("please", 4);  
multiprint("beetlejuice", 3);
```

You would expect these calls to produce the following output:

```
[please, please, please, please]  
[beetlejuice, beetlejuice, beetlejuice]
```

Your first attempt at writing this method might be a simple loop that prints square brackets outside the loop and prints the string and a comma inside the loop:

```
public static void multiprint(String s, int times) {  
    System.out.print("[");  
    for (int i = 1; i <= times; i++) {  
        System.out.print(s + ", ");  
    }  
    System.out.println("]");  
}
```

Unfortunately, this code produces an extraneous comma after the last value:

```
[please, please, please, please, ]  
[beetlejuice, beetlejuice, beetlejuice, ]
```

Because the commas are separators, you want to print one more string than comma (e.g., two commas to separate the three occurrences of “beetlejuice”). You can use the classic solution to the fencepost problem to achieve this effect by printing one string outside the loop and reversing the order of the printing inside the loop:

```
public static void multiprint(String s, int times) {
    System.out.print("[ " + s);
    for (int i = 2; i <= times; i++) {
        System.out.print(", " + s);
    }
    System.out.println("]");
}
```

Notice that because you’re printing one of the strings before the loop begins, you have to modify the loop so that it won’t print as many strings as it did before. Adjusting the loop variable *i* to start at 2 accounts for the first value that is printed before the loop.

Unfortunately, this solution does not work properly either. Consider what happens when you ask the method to print a string zero times, as in:

```
multiprint("please don't", 0);
```

This call produces the following incorrect output:

```
[please don't]
```

You want it to be possible for a user to request zero occurrences of a string, so the method shouldn’t produce that incorrect output. The problem is that the classic solution to the fencepost problem involves printing one value before the loop begins. To get the method to behave correctly for the zero case, you can include an *if/else* statement:

```
public static void multiprint(String s, int times) {
    if (times == 0) {
        System.out.println("[ ]");
    } else {
        System.out.print("[ " + s);
        for (int i = 2; i <= times; i++) {
            System.out.print(", " + s);
        }
        System.out.println("]");
    }
}
```

Alternatively, you can include an `if` statement inside the loop (the double-test approach):

```
public static void multiprint(String s, int times) {
    System.out.print("[");
    for (int i = 1; i <= times; i++) {
        System.out.print(s);
        if (i < times) {
            System.out.print(", ");
        }
    }
    System.out.println("]");
}
```

Although the preceding version of the code performs a similar test twice on each iteration, it is simpler than using the classic fencepost solution and its special case. Neither solution is better than the other, as there is a tradeoff involved. If you think that the code will be executed often and that the loop will iterate many times, you might be more inclined to use the efficient solution. Otherwise, you might choose the simpler code.

5.3 The boolean Type



VideoNote

George Boole was such a good logician that Java has a data type named after him. The Java type `boolean` is used to describe logical true/false relationships. Recall that `boolean` is one of the primitive types, like `int`, `double`, and `char`.

Novices often wonder why computer scientists are so interested in logic. The answer is that logic is fundamental to computing in the same way that physics is fundamental to engineering. Engineers study physics because they want to build real-world artifacts that are governed by the laws of physics. If you don't understand physics, you're likely to build a bridge that will collapse. Computer scientists build artifacts as well, but in a virtual world that is governed by the laws of logic. If you don't understand logic, you're likely to build computer programs that collapse.

Without realizing it, you have already used `booleans`. All of the control structures we have looked at—`if/else` statements, `for` loops, and `while` loops—are controlled by expressions that specify tests. For example, the expression

```
number % 2 == 0
```

is a test for divisibility by 2. It is also a Boolean expression. Boolean expressions are meant to capture the concepts of truth and falsity, so it is not surprising that the

domain of type `boolean` has only two values: `true` and `false`. The words `true` and `false` are reserved words in Java. They are the literal values of type `boolean`. All Boolean expressions, when evaluated, will return one or the other of these literals. Don't confuse these special values with the string literals `"true"` and `"false"`. You don't need quotes to refer to the Boolean literals.

When you write a program that manipulates numerical values, you'll often want to compute a value and store it in a variable or write a method that captures some complex formula. We end up doing the same thing with type `boolean`. We might want to record a Boolean value in a variable and we often write methods that return Boolean results. For example, we have seen that the `String` class has methods that return a Boolean result, including `startsWith`, `endsWith`, `equals`, and `equalsIgnoreCase`.

To understand this better, remember what these terms mean for the type `int`. The literals of type `int` include 0, 1, 2, and so on. Because these are literals of type `int`, you can write expressions like the following ones with them:

```
int number1 = 1;
int number2 = 0;
```

Consider what you can do with variables of type `boolean`. Suppose you define two boolean variables, `test1` and `test2`. These variables can take on only two possible values: `true` and `false`. You can say:

```
boolean test1 = true;
boolean test2 = false;
```

You can also write a statement that copies the value of one `boolean` variable to another, as with variables of any other type:

```
test1 = test2;
```

Furthermore, you know that the assignment statement can use expressions like:

```
number1 = 2 + 2;
```

and that the simple tests you have been using are Boolean expressions. That means you can write statements like the following ones:

```
test1 = (2 + 2 == 4);
test2 = (3 * 100 < 250);
```

These assignment statements say, in effect, “Set this `boolean` variable according to the truth value returned by the following test.” The first statement sets the variable `test1` to `true`, because the test evaluates to `true`. The second sets the variable `test2` to `false`, because the second test evaluates to `false`. You don’t need to include parentheses, but they make the statements more readable.

Obviously, then, assignment is one of the operations you can perform on variables of type `boolean`.

Logical Operators

In Java, you can form complicated Boolean expressions using what are known as the *logical operators*, shown in Table 5.3.

Table 5.3 Logical Operators

Operator	Meaning	Example	Value
<code>&&</code>	AND (conjunction)	<code>(2 == 2) && (3 < 4)</code>	<code>true</code>
<code> </code>	OR (disjunction)	<code>(1 < 2) (2 == 3)</code>	<code>true</code>
<code>!</code>	NOT (negation)	<code>!(2 == 2)</code>	<code>false</code>

The NOT operator (`!`) reverses the truth value of its operand. If an expression evaluates to `true`, its negation evaluates to `false`, and vice versa. You can express this relationship in a truth table. The truth table that follows has two columns, one for a variable and one for its negation. For each value of the variable, the table shows the corresponding value of the negation.

Truth Table for NOT (!)

P	!p
<code>true</code>	<code>false</code>
<code>false</code>	<code>true</code>

In addition to the negation operator, there are two logical connectives you will use, AND (`&&`) and OR (`||`). You use these connectives to tie together two Boolean expressions, creating a new Boolean expression. The following truth table shows that the AND operator evaluates to `true` only when both of its individual operands are `true`.

Truth Table for AND (&&)

P	q	p && q
<code>true</code>	<code>true</code>	<code>true</code>
<code>true</code>	<code>false</code>	<code>false</code>
<code>false</code>	<code>true</code>	<code>false</code>
<code>false</code>	<code>false</code>	<code>false</code>

The following truth table shows that the OR operator evaluates to true except when both operands are false.

Truth Table for OR (||)

p	q	p q
true	true	true
true	false	true
false	true	true
false	false	false

The Java OR operator has a slightly different meaning from the English word “or.” In English you say, “I’ll study tonight or I’ll go to a movie.” One or the other will be true, but not both. The OR operator is more like the English expression “and/or”: If one or both operands are true, the overall proposition is true.

You generally use logical operators when what you have to say cannot be reduced to a single test. For example, as we saw in the previous chapter, if you want to do a particular operation when a number is between 1 and 10, you might say,

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

But you can say this more easily using logical AND:

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

People use the words “and” and “or” all the time, but Java only allows you to use them in the strict logical sense. Be careful not to write code like the following:

 `// this does not compile`

```
if (x == 1 || 2 || 3) {
    doSomething();
}
```

In English, we would read this as “x equals 1 or 2 or 3,” which makes sense to us, but it doesn’t make sense to Java. You might also be tempted to write code like the following:

 `// this does not compile`

```
if (1 <= x <= 10) {
    doSomethingElse();
}
```

In mathematics, this expression would make sense and would test whether *x* is between 1 and 10 inclusive. However, the expression doesn't make sense in Java.

You can only use the logical AND and OR operators to combine a series of Boolean expressions. Otherwise, the computer will not understand what you mean. To express the “1 or 2 or 3” idea, combine three different Boolean expressions with logical ORs:

```
if (x == 1 || x == 2 || x == 3) {
    doSomething();
}
```

To express the “between 1 and 10 inclusive” idea, combine two Boolean expressions with a logical AND:

```
if (1 <= x && x <= 10) {
    doSomethingElse();
}
```

Now that we've introduced the AND, OR, and NOT logical operators, it's time to revisit our precedence table. The NOT operator appears at the top, with the highest level of precedence. The other two logical operators have fairly low precedence, lower than the arithmetic and relational operators but higher than the assignment operators. The AND operator has a slightly higher level of precedence than the OR operator. Table 5.4 includes these new operators.

According to these rules of precedence, when Java evaluates an expression like the following one, the computer will evaluate the NOT first, the AND second, and then the OR.

```
if (test1 || !test2 && test3) {
    doSomething();
}
```

Table 5.4 Java Operator Precedence

Description	Operators
unary operators	!, ++, --, +, -
multiplicative operators	*, /, %
additive operators	+, -
relational operators	<, >, <=, >=
equality operators	==, !=
logical AND	&&
logical OR	
assignment operators	=, +=, -=, *=, /=, %=, &&=, =

Short-Circuited Evaluation

In this section we will explore the use of the logical operators to solve a complex programming task, and we'll introduce an important property of these operators. We will write a method called `firstWord` that takes a `String` as a parameter and returns the first word in the string. To keep things simple, we will adopt the convention that a `String` is broken up into individual words by spaces. If the `String` has no words at all, the method should return an empty string. Here are a few example calls:

Method Call	Value Returned
<code>firstWord("four score and seven years")</code>	"four"
<code>firstWord("all-one-word-here")</code>	"all-one-word-here"
<code>firstWord(" lots of space here")</code>	"lots"
<code>firstWord(" ")</code>	" "

Remember that we can call the `substring` method to pull out part of a string. We pass two parameters to the `substring` method: the starting index of the substring and the index one beyond the end of the substring. If the string is stored in a variable called `s`, our task basically reduces to the following steps:

```
set start to the first index of the word.
set stop to the index just beyond the word.
return s.substring(start, stop).
```

As a first approximation, let's assume that the starting index is 0. This starting index won't work for strings that begin with spaces, but it will allow us to focus on the second step in the pseudocode. Consider a string that begins with "four score". If we examine the individual characters of the string and their indexes, we find the following pattern:

0	1	2	3	4	5	6	7	8	9	...
f	o	u	r		s	c	o	r	e	...

We set `start` to 0. We want to set the variable `stop` to the index just beyond the end of the first word. In this example, the word we want is "four", and it extends from indexes 0 through 3. So, if we want the variable `stop` to be one beyond the end of the desired substring, we want to set it to index 4, the location of the first space in the string.

So how do we find the first space in the string? We use a `while` loop. We simply start at the front of the string and loop until we get to a space:

```
set stop to 0.
while (the character at index stop is not a space) {
    increase stop by 1.
}
```


This is easily converted into Java code. Combining it with our assumption that `start` will be 0, we get:

```
public static String firstWord(String s) {
    int start = 0;
    int stop = 0;
    while (s.charAt(stop) != ' ') {
        stop++;
    }
    return s.substring(start, stop);
}
```

This version of the method works for many cases, including our sample string, but it doesn't work for all strings. It has two major limitations. We began by assuming that the string did not begin with spaces, so we know we have to fix that limitation.

The second problem is that this version of `firstWord` doesn't work on one-word strings. For example, if we execute it with a string like "four", it generates a `StringIndexOutOfBoundsException` indicating that 4 is not a legal index. The exception occurs because our code assumes that we will eventually find a space, but there is no space in the string "four". So `stop` is incremented until it becomes equal to 4, and an exception is thrown because there is no character at index 4. This is sometimes referred to as "running off the end of the string."

To address this problem, we need to incorporate a test that involves the length of the string. Many novices attempt to do this by using some combination of `while` and `if`, as in the following code:

```
int stop = 0;
while (stop < s.length()) {
    if (s.charAt(stop) != ' ') {
        stop++;
    }
}
```

This code works for one-word strings like "four" because as soon as `stop` becomes equal to the length of the string, we break out of the loop. However, it doesn't work for the original multiword cases like "four score". We end up in an infinite loop because once `stop` becomes equal to 4, we stop incrementing it, but we get trapped inside the loop because the test says to continue as long as `stop` is less than the length of the string. This approach of putting an `if` inside a `while` led to a world-famous bug on December 31, 2008, when Zune music players all over the world stopped working (see Self-Check Problem 21 for more details).

The point to recognize is that in this case we need to use two different conditions in controlling the loop. We want to continue incrementing `stop` only if we know that

we haven't seen a space *and* that we haven't reached the end of the string. We can express that idea using the logical AND operator:

```
int stop = 0;
while (s.charAt(stop) != ' ' && stop < s.length()) {
    stop++;
}
```

Unfortunately, even this test does not work. It expresses the two conditions properly, because we want to make sure that we haven't reached a space and we want to make sure that we haven't reached the end of the string. But think about what happens just as we reach the end of a string. Suppose that `s` is "four" and `stop` is equal to 3. We see that the character at index 3 is not a space and we see that `stop` is less than the length of the string, so we increment one more time and `stop` becomes 4. As we come around the loop, we test whether `s.charAt(4)` is a space. This test throws an exception. We also test whether `stop` is less than 4, which it isn't, but that test comes too late to avoid the exception.

Java offers a solution for this situation. The logical operators `&&` and `||` use *short-circuited evaluation*.

Short-Circuited Evaluation

The property of the logical operators `&&` and `||` that prevents the second operand from being evaluated if the overall result is obvious from the value of the first operand.

In our case, we are performing two different tests and asking for the logical AND of the two tests. If either test fails, the overall result is `false`, so if the first test fails, it's not necessary to perform the second test. Because of short-circuited evaluation—that is, because the overall result is obvious from the first test—we don't perform the second test at all. In other words, the performance and evaluation of the second test are prevented (short-circuited) by the fact that the first test fails.

This means we need to reverse the order of our two tests:

```
int stop = 0;
while (stop < s.length() && s.charAt(stop) != ' ') {
    stop++;
}
```

If we run through the same scenario again with `stop` equal to 3, we pass both of these tests and increment `stop` to 4. Then, as we come around the loop again, we first test to see if `stop` is less than `s.length()`. It is not, which means the test evaluates to `false`. As a result, Java knows that the overall expression will evaluate to `false` and never evaluates the second test. This order of events prevents the exception from occurring, because we never test whether `s.charAt(4)` is a space.

This solution gives us a second version of the method:

```
public static String firstWord(String s) {  
    int start = 0;  
    int stop = 0;  
    while (stop < s.length() && s.charAt(stop) != ' ') {  
        stop++;  
    }  
    return s.substring(start, stop);  
}
```

But remember that we assumed that the first word starts at position 0. That won't necessarily be the case. For example, if we pass a string that begins with several spaces, this method will return an empty string. We need to modify the code so that it skips any leading spaces. Accomplishing that goal requires another loop. As a first approximation, we can write the following code:

```
int start = 0;  
while (s.charAt(start) == ' ') {  
    start++;  
}
```

This code works for most strings, but it fails in two important cases. The loop test assumes we will find a nonspace character. What if the string is composed entirely of spaces? In that case, we'll simply run off the end of the string, generating a `StringIndexOutOfBoundsException`. And what if the string is empty to begin with? We'll get an error immediately when we ask about `s.charAt(0)`, because there is no character at index 0.

We could decide that these cases constitute errors. After all, how can you return the first word if there is no word? So, we could document a precondition that the string contains at least one nonspace character, and throw an exception if we find that it doesn't. Another approach is to return an empty string in these cases.

To deal with the possibility of the string being empty, we need to modify our loop to incorporate a test on the length of the string. If we add it at the end of our while loop test, we get the following code:

```
 int start = 0;  
while (s.charAt(start) == ' ' && start < s.length()) {  
    start++;  
}
```

But this code has the same flaw we saw before. It is supposed to prevent problems when `start` becomes equal to the length of the string, but when this situation occurs, a `StringIndexOutOfBoundsException` will be thrown before the computer reaches

the test on the length of the string. So these tests also have to be reversed to take advantage of short-circuited evaluation:

```
int start = 0;
while (start < s.length() && s.charAt(start) == ' ') {
    start++;
}
```

To combine these lines of code with our previous code, we have to change the initialization of `stop`. We no longer want to search from the front of the string. Instead, we need to initialize `stop` to be equal to `start`. Putting these pieces together, we get the following version of the method:

```
public static String firstWord(String s) {
    int start = 0;
    while (start < s.length() && s.charAt(start) == ' ') {
        start++;
    }
    int stop = start;
    while (stop < s.length() && s.charAt(stop) != ' ') {
        stop++;
    }
    return s.substring(start, stop);
}
```

This version works in all cases, skipping any leading spaces and returning an empty string if there is no word to return.

boolean Variables and Flags

All `if/else` statements are controlled by Boolean tests. The tests can be boolean variables or Boolean expressions. Consider, for example, the following code:

```
if (number > 0) {
    System.out.println("positive");
} else {
    System.out.println("not positive");
}
```

This code could be rewritten as follows:

```
boolean positive = (number > 0);
if (positive) {
    System.out.println("positive");
} else {
    System.out.println("not positive");
}
```

Using boolean variables adds to the readability of your programs because it allows you to give names to tests. Consider the kind of code you would generate for a dating program. You might have some integer variables that describe certain attributes of a person: `looks`, to store a rough estimate of physical beauty (on a scale of 1–10); `IQ`, to store intelligence quotient; `income`, to store gross annual income; and `snothers`, to track intimate friends (“snother” is short for “significant other”). Given these variables to specify a person’s attributes, you can develop various tests of suitability. As you are writing the program, you can use boolean variables to give names to those tests, adding greatly to the readability of the code:

```
boolean cute = (looks >= 9);
boolean smart = (IQ > 125);
boolean rich = (income > 100000);
boolean available = (snothers == 0);
boolean awesome = cute && smart && rich && available;
```

You might find occasion to use a special kind of boolean variable called a *flag*. Typically we use flags within loops to record error conditions or to signal completion. Different flags test different conditions. As an analogy, consider a referee at a sports game who watches for a particular illegal action and throws a flag if it happens. You sometimes hear an announcer saying, “There is a flag down on the play.”

Let’s introduce a flag into the cumulative sum code we saw in the previous chapter:

```
double sum = 0.0;
for (int i = 1; i <= totalNumber; i++) {
    System.out.print("    #" + i + "? ");
    double next = console.nextDouble();
    sum += next;
}
System.out.println("sum = " + sum);
```

Suppose we want to know whether the sum ever goes negative at any point. Notice that this situation isn’t the same as the situation in which the sum ends up being negative. Like a bank account balance, the sum might switch back and forth between positive and negative. As you make a series of deposits and withdrawals, the bank will keep track of whether you overdraw your account along the way. Using a boolean flag, we can modify the preceding loop to keep track of whether the sum ever goes negative and report the result after the loop:

```
double sum = 0.0;
boolean negative = false;
for (int i = 1; i <= totalNumber; i++) {
    System.out.print("    #" + i + "? ");
    double next = console.nextDouble();
```



```

        sum += next;
    if (sum < 0.0) {
        negative = true;
    }
}
System.out.println("sum = " + sum);
if (negative) {
    System.out.println("Sum went negative");
} else {
    System.out.println("Sum never went negative");
}

```

Boolean Zen

In 1974, Robert Pirsig started a cultural trend with his book *Zen and the Art of Motorcycle Maintenance: An Inquiry into Values*. A slew of later books copied the title with *Zen and the Art of X*, where *X* was Poker, Knitting, Writing, Foosball, Guitar, Public School Teaching, Making a Living, Falling in Love, Quilting, Stand-up Comedy, the SAT, Flower Arrangement, Fly Tying, Systems Analysis, Fatherhood, Screenwriting, Diabetes Maintenance, Intimacy, Helping, Street Fighting, Murder, and on and on. There was even a book called *Zen and the Art of Anything*.

We now join this cultural trend by discussing Zen and the art of type `boolean`. It seems to take a while for many novices to get used to Boolean expressions. Novices often write overly complex expressions involving `boolean` values because they don't grasp the simplicity that is possible when you "get" how the `boolean` type works.

For example, suppose that you are writing a game-playing program that involves two-digit numbers, each of which is composed of two different digits. In other words, the program will use numbers like 42 that are composed of two distinct digits, but not numbers like 6 (only one digit), 394 (more than two digits), or 22 (both digits are the same). You might find yourself wanting to test whether a given number is legal for use in the game. You can restrict yourself to two-digit numbers with a test like the following one:

```
n >= 10 && n <= 99
```

You also have to test to make sure that the two digits aren't the same. You can get the digits of a two-digit number with the expressions `n / 10` and `n % 10`. So you can expand the test to ensure that the digits aren't the same:

```
n >= 10 && n <= 99 && (n / 10 != n % 10)
```

This test is a good example of a situation in which you could use a method to capture a complex Boolean expression. Returning a `boolean` will allow you to call the method as many times as you want without having to copy this complex expression each time, and you can give a name to this computation to make the program more readable.

Suppose you want to call the method `isTwoUniqueDigits`. You want the method to take a value of type `int` and return `true` if the `int` is composed of two unique digits and `false` if it is not. So, the method would look like the following:

```
public static boolean isTwoUniqueDigits(int n) {
    ...
}
```

How would you write the body of this method? We've already written the test, so we just have to figure out how to incorporate it into the method. The method has a boolean return type, so you want it to return the value `true` when the test succeeds and the value `false` when it fails. You can write the method as follows:

```
public static boolean isTwoUniqueDigits(int n) {
    if (n >= 10 && n <= 99 && (n % 10 != n / 10)) {
        return true;
    } else {
        return false;
    }
}
```

This method works, but it is more verbose than it needs to be. The preceding code evaluates the test that we developed. That expression is of type `boolean`, which means that it evaluates to either `true` or `false`. The `if/else` statement tells the computer to return `true` if the expression evaluates to `true` and to return `false` if it evaluates to `false`. But why use this construct? If the method is going to return `true` when the expression evaluates to `true` and return `false` when it evaluates to `false`, you can just return the value of the expression directly:

```
public static boolean isTwoUniqueDigits(int n) {
    return (n >= 10 && n <= 99 && (n % 10 != n / 10));
}
```

Even the preceding version can be simplified, because the parentheses are not necessary (although they make it clearer exactly what the method will return). This code evaluates the test that we developed to determine whether a number is composed of two unique digits and returns the result (`true` when it does, `false` when it does not).

Consider an analogy to integer expressions. To someone who understands Boolean Zen, the `if/else` version of this method looks as odd as the following code:

```
if (x == 1) {
    return 1;
} else if (x == 2) {
    return 2;
} else if (x == 3) {
    return 3;
}
```

```

} else if (x == 4) {
    return 4;
} else if (x == 5) {
    return 5;
}

```

If you always want to return the value of `x`, you should just say:

```
return x;
```

A similar confusion can occur when students use `boolean` variables. In the last section we looked at a variation of the cumulative sum algorithm that used a `boolean` variable called `negative` to keep track of whether or not the sum ever goes negative. We then used an `if/else` statement to print a message reporting the result:

```

if (negative) {
    System.out.println("Sum went negative");
} else {
    System.out.println("Sum never went negative");
}

```

Some novices would write this code as follows:

```

if (negative == true) {
    System.out.println("Sum went negative");
} else {
    System.out.println("Sum never went negative");
}

```

The comparison is unnecessary because the `if/else` statement expects an expression of type `boolean` to appear inside the parentheses. A `boolean` variable is already of the appropriate type, so we don't need to test whether it equals `true`; it either *is* `true` or it isn't (in which case it is `false`). To someone who understands Boolean Zen, the preceding test seems as redundant as saying:

```

if ((negative == true) == true) {
    ...
}

```

Novices also often write tests like the following:

```

if (negative == false) {
    ...
}

```

This makes some sense because the test is doing something useful, in that it switches the meaning of the boolean variable (evaluating to `true` if the variable is `false` and evaluating to `false` if the variable is `true`). But the negation operator is designed to do this kind of switching of boolean values, so this test is better written as follows:

```
if (!negative) {  
    ...  
}
```

You should get used to reading the exclamation mark as “not”, so this test would be read as “if not negative.” To those who understand Boolean Zen, that is a more concise way to express the test than to test whether `negative` is equal to `false`.

Negating Boolean Expressions

Programmers often find themselves needing to form the negation of a complex Boolean expression. For example, it is often easiest to reason about a loop in terms of an exit condition that would make us want to stop the loop, but the `while` loop requires us to express the code in terms of a continuation condition. Suppose that you want to write a loop that keeps prompting the user for an integer until that integer is a two-digit number. Because two-digit numbers range from 10 to 99, you can use the following lines of code to test whether a number has exactly two digits:

```
number >= 10 && number <= 99
```

To put this in a `while` loop, we have to turn the test around because the `while` loop test is a continuation test. In other words, we want to stay in the loop while this is *not* true (while the user has not yet given us a two-digit number). One approach is to use the logical NOT operator to negate this expression:

```
while (!(number >= 10 && number <= 99))
```

Notice that we need to parenthesize the entire Boolean expression and then put the NOT operator in front of it. While this approach works, it is generally considered bad style. It is best to simplify this expression.

A general approach to simplifying such expressions was formalized by the British logician Augustus De Morgan. We can apply one of two rules that are known as *De Morgan's laws*.

Table 5.5 shows De Morgan's laws. Notice that when you negate a Boolean expression, each operand is negated (`p` becomes `!p` and `q` becomes `!q`) and the logical operator flips. Logical OR becomes logical AND and vice versa when you compute the negation.

Table 5.5 De Morgan's Laws

Original expression	Negated expression	Simplified negation
<code>p q</code>	<code>!(p q)</code>	<code>!p && !q</code>
<code>p && q</code>	<code>!(p && q)</code>	<code>!p !q</code>

We can use the first De Morgan's law for our two-digit number program because we are trying to find the negation of an expression that involves the logical OR operator. Instead of writing:

```
while (!(number >= 10 && number <= 99))
```

we can say:

```
while (number < 10 || number > 99)
```

Each individual test has been negated and the AND operator has been replaced with an OR operator.

Let's look at a second example that involves the other De Morgan's law. Suppose that you want to ask the user a question and you want to force the user to answer either "yes" or "no". If you have a `String` variable called `response`, you can use the following test to describe what you want to be true:

```
response.equals("yes") || response.equals("no")
```

If we're writing a loop to keep reading a response until this expression evaluates to true, then we want to write the loop so that it uses the negation of this test. So, once again, we could use the NOT operator for the entire expression:

```
while (!(response.equals("yes") || response.equals("no")))
```

Once again it is best to simplify the expression using De Morgan's law:

```
while (!response.equals("yes") && !response.equals("no"))
```

5.4 User Errors

In the previous chapter, you learned that it is good programming practice to think about the preconditions of a method and to mention them in the comments for the method. You also learned that in some cases your code can throw exceptions if preconditions are violated.

When you are writing interactive programs, the simplest approach is to assume that the user will provide good input. You can then document your preconditions and throw exceptions when the user input isn't what was expected. In general, though, it's better to write programs that don't make assumptions about user input. You've seen, for example, that the `Scanner` object can throw an exception if the user enters the wrong kind of data. It's preferable to write programs that can deal with user errors. Such programs are referred to as being *robust*.

Robust

Ability of a program to execute even when presented with illegal data.

In this section we will explore how to write robust interactive programs. Before you can write robust code, though, you have to understand some special functionality of the `Scanner` class.

Scanner Lookahead

The `Scanner` class has methods that allow you to perform a test before you read a value. In other words, it allows you to look before you leap. For each of the “next” methods of the `Scanner` class, there is a corresponding “has” method that tells you whether or not you can perform the given operation.

For example, you will often want to read an `int` using a `Scanner` object. But what if the user types something other than an `int`? `Scanner` has a method called `hasNextInt` that tells you whether or not reading an `int` is currently possible. To determine whether it is possible, the `Scanner` object looks at the next token and checks whether it can be interpreted as an integer.

We tend to interpret certain sequences of characters as particular types of data, but when we read tokens, they can be interpreted in different ways. The following program will allow us to explore this concept:

```
1 import java.util.*;
2
3 public class ExamineInput1 {
4     public static void main(String[] args) {
5         System.out.println("This program examines the ways");
6         System.out.println("a token can be read.");
7         System.out.println();
8
9         Scanner console = new Scanner(System.in);
10
11         System.out.print("token? ");
12         System.out.println("  hasNextInt = " +
13                             console.hasNextInt());
14         System.out.println("  hasNextDouble = " +
15                             console.hasNextDouble());
```

```
16         System.out.println("  hasNext = " + console.hasNext());
17     }
18 }
```

Let's look at a few sample executions. Here is the output of the program when we enter the token 348:

```
This program examines the ways
a token can be read.
```

```
token? 348
  hasNextInt = true
  hasNextDouble = true
  hasNext = true
```

As you'd expect, the call on `hasNextInt` returns `true`, which means that we could interpret this token as an integer. The `Scanner` would also allow us to interpret this token as a double, so `hasNextDouble` also returns `true`. But notice that `hasNext()` returns `true` as well. That result means that we could call the `next` method to read in this token as a `String`.

Here's another execution, this time for the token 348.2:

```
This program examines the ways
a token can be read.
```

```
token? 348.2
  hasNextInt = false
  hasNextDouble = true
  hasNext = true
```

This token cannot be interpreted as an `int`, but it can be interpreted as a double or a `String`. Finally, consider this execution for the token `hello`:

```
This program examines the ways
a token can be read.
```

```
token? hello
  hasNextInt = false
  hasNextDouble = false
  hasNext = true
```

The token `hello` can't be interpreted as an `int` or double; it can only be interpreted as a `String`.

Handling User Errors

Consider the following code fragment:

```
Scanner console = new Scanner(System.in);
System.out.print("How old are you? ");
int age = console.nextInt();
```

What if the user types something that is not an integer? If that happens, the `Scanner` will throw an exception on the call to `nextInt`. We saw in the previous section that we can test whether or not the next token can be interpreted as an integer by using the `hasNextInt` method. So, we can test before reading an `int` whether the user has typed an appropriate value.

If the user types something other than an integer, we want to discard the input, print out some kind of error message, and prompt for a second input. We want this code to execute in a loop so that we keep discarding input and generating error messages as necessary until the user enters legal input.

Here is a first attempt at a solution in pseudocode:

```
while (user hasn't given us an integer) {
    prompt.
    discard input.
    generate an error message.
}
read the integer.
```

This reflects what we want to do, in general. We want to keep prompting, discarding, and generating error messages as long as the input is illegal, and when a legal value is entered, we want to read the integer. Of course, in that final case we don't want to discard the input or generate an error message. In other words, the last time through the loop we want to do just the first of these three steps (prompting, but not discarding and not generating an error message). This is another classic fencepost problem, and we can solve it in the usual way by putting the initial prompt before the loop and changing the order of the operations within the loop:

```
prompt.
while (user hasn't given us an integer) {
    discard input.
    generate an error message.
    prompt.
}
read the integer.
```

This pseudocode is fairly easy to turn into actual Java code:

```
Scanner console = new Scanner(System.in);
System.out.print("How old are you? ");
while (!console.hasNextInt()) {
    console.next(); // to discard the input
    System.out.println("Not an integer; try again.");
    System.out.print("How old are you? ");
}
int age = console.nextInt();
```

In fact, this is such a common operation that it is worth turning into a static method:

```
// prompts until a valid number is entered
public static int getInt(Scanner console, String prompt) {
    System.out.print(prompt);
    while (!console.hasNextInt()) {
        console.next(); // to discard the input
        System.out.println("Not an integer; try again.");
        System.out.print(prompt);
    }
    return console.nextInt();
}
```

Using this method, we can rewrite our original code as follows:

```
Scanner console = new Scanner(System.in);
int age = getInt(console, "How old are you? ");
```

When you execute this code, the interaction looks like this:

```
How old are you? what?
Not an integer; try again.
How old are you? 18.4
Not an integer; try again.
How old are you? ten
Not an integer; try again.
How old are you? darn!
Not an integer; try again.
How old are you? help
Not an integer; try again.
How old are you? 19
```

5.5 Assertions and Program Logic

Logicians concern themselves with declarative statements called *assertions*.

Assertion

A declarative sentence that is either true or false.

The following statements are all assertions:

- $2 + 2$ equals 4.
- The sun is larger than the Earth.
- $x > 45$.
- It was raining.
- The rain in Spain falls mainly on the plain.

The following statements are not assertions (the first is a question and the second is a command):

- How much do you weigh?
- Take me home.

Some assertions are true or false depending upon their context:

- $x > 45$. (The validity of this statement depends on the value of x .)
- It was raining. (The validity of this statement depends on the time and location.)

You can pin down whether they are true or false by providing a context:

- When $x = 13$, $x > 45$.
- On July 4, 1776, in Philadelphia, it was raining.

To write programs correctly and efficiently, you must learn to make assertions about your programs and to understand the contexts in which those assertions will be true. For example, if you are trying to obtain a nonnegative number from the user, you want the assertion “Number is nonnegative” to be true. You can use a simple prompt and read:

```
System.out.print("Please give me a nonnegative number--> ");
double number = console.nextDouble();
// is number nonnegative?
```

But the user can ignore your request and input a negative number anyway. In fact, users often input values that you don't expect, usually because they are confused. Given

the uncertainty of user input, this particular assertion may sometimes be true and sometimes false. But something later in the program may depend on the assertion being true. For example, if you are going to take the square root of that number, you must be sure the number is nonnegative. Otherwise, you might end up with a bad result.

Using a loop, you can guarantee that the number you get is nonnegative:

```
System.out.print("Please give me a nonnegative number--> ");
double number = console.nextDouble();
while (number < 0.0) {
    System.out.print("That is a negative number. Try again--> ");
    number = console.nextDouble();
}
// is number nonnegative?
```

You know that `number` will be nonnegative after the loop; otherwise, the program would not exit the `while` loop. As long as a user gives negative values, your program stays in the `while` loop and continues to prompt for input.

This doesn't mean that `number` *should* be nonnegative after the loop. It means that `number` *will* be nonnegative. By working through the logic of the program, you can see that this is a certainty, an assertion of which you are sure. You could even prove it if need be. Such an assertion is called a *provable assertion*.

Provable Assertion

An assertion that can be proven to be true at a particular point in program execution.

Provable assertions help to identify unnecessary bits of code. Consider the following statements:

```
int x = 0;
if (x == 0) {
    System.out.println("This is what I expect.");
} else {
    System.out.println("How can that be?");
}
```

The `if/else` construct is not necessary. You know what the assignment statement does, so you know that it sets `x` to 0. Testing whether `x` is 0 is as unnecessary as saying, "Before I proceed, I'm going to check that $2 + 2$ equals 4." Because the `if` part of this `if/else` statement is always executed, you can prove that the following lines of code always do the same thing as the preceding lines:

```
int x = 0;
System.out.println("This is what I expect.");
```

This code is simpler and, therefore, better. Programs are complex enough without adding unnecessary code.

The concept of assertions has become so popular among software practitioners that many programming languages provide support for testing assertions. Java added support for testing assertions starting with version 1.4 of the language. You can read more about Java's `assert` statement in Appendix C.

Reasoning about Assertions

The focus on assertions comes out of a field of computer science known as *formal verification*.

Formal Verification

A field of computer science that involves reasoning about the formal properties of programs to prove the correctness of a program.

For example, consider the properties of the simple `if` statement:

```
if (<test>) {  
    // test is always true here  
    ...  
}
```

You enter the body of the `if` statement only if the test is true, which is why you know that the test must be true if that particular line is reached in program execution. You can draw a similar conclusion about what is true in an `if/else` statement:

```
if (<test>) {  
    // test is always true here  
    ...  
} else {  
    // test is never true here  
    ...  
}
```

You can draw a similar conclusion about what is true inside the body of a `while` loop:

```
while (<test>) {  
    // test is always true here  
    ...  
}
```

But in the case of the `while` loop, you can draw an even stronger conclusion. You know that as long as the test evaluates to true, you'll keep going back into the loop.

Thus, you can conclude that after the loop is done executing, the test can no longer be true:

```
while (<test>) {
    // test is always true here
    ...
}
// test is never true here
```

The test can't be true after the loop because if it had been true, the program would have executed the body of the loop again.

These observations about the properties of `if` statements, `if/else` statements, and `while` loops provide a good start for proving certain assertions about programs. But often, proving assertions requires a deeper analysis of what the code actually does. For example, suppose you have a variable `x` of type `int` and you execute the following `if` statement:

```
if (x < 0) {
    // x < 0 is always true here
    x = -x;
}
// but what about x < 0 here?
```

You wouldn't normally be able to conclude anything about `x` being less than 0 after the `if` statement, but you can draw a conclusion if you think about the different cases. If `x` was greater than or equal to 0 before the `if` statement, it will still be greater than or equal to 0 after the `if` statement. And if `x` was less than 0 before the `if` statement, it will be equal to $-x$ after. When `x` is less than 0, $-x$ is greater than 0. Thus, in either case, you know that after the `if` statement executes, `x` will be greater than or equal to 0.

Programmers naturally apply this kind of reasoning when writing programs. Computer scientists are trying to figure out how to do this kind of reasoning in a formal, verifiable way.

A Detailed Assertions Example



To explore assertions further, let's take a detailed look at a code fragment and a set of assertions we might make about the fragment. Consider the following method:

```
public static void printCommonPrefix(int x, int y) {
    int z = 0;
    // Point A
    while (x != y) {
        // Point B
        z++;
    }
}
```

```

        // Point C
        if (x > y) {
            // Point D
            x = x / 10;
        } else {
            // Point E
            y = y / 10;
        }
        // Point F
    }
    // Point G
    System.out.println("common prefix = " + x);
    System.out.println("digits discarded = " + z);
}

```

This method finds the longest sequence of leading digits that two numbers have in common. For example, the numbers 32845 and 328929343 each begin with the prefix 328. This method will report that prefix and will also report the total number of digits that follow the common prefix and that are discarded.

We will examine the program to check whether various assertions are always true, never true, or sometimes true and sometimes false at various points in program execution. The comments in the method indicate the points of interest. The assertions we will consider are:

```

x > y
x == y
z == 0

```

Normally computer scientists write assertions in mathematical notation, as in $z = 0$, but we will use a Java expression to distinguish this assertion of equality from the practice of assigning a value to the variable.

We can record our answers in a table with the words “always,” “never,” or “sometimes.” Our table will look like the following one:

	$x > y$	$x == y$	$z == 0$
Point A			
Point B			

Let’s start at point A, which appears near the beginning of the method’s execution:

```

public static void printCommonPrefix(int x, int y) {
    int z = 0;
    // Point A

```

The variables `x` and `y` are parameters and get their values from the call to the method. Many calls are possible, so we don't really know anything about the values of `x` and `y`. Thus, the assertion `x > y` could be true but doesn't have to be. The assertion is sometimes true, sometimes false at point A. Likewise, the assertion `x == y` could be true depending on what values are passed to the method, but it doesn't have to be true. However, we initialize the local variable `z` to 0 just before point A, so the assertion `z == 0` will always be true at that point in execution. So, we can fill in the first line of the table as follows:

	<code>x > y</code>	<code>x == y</code>	<code>z == 0</code>
Point A	sometimes	sometimes	always

Point B appears just inside the `while` loop:

```
while (x != y) {
    // Point B
    z++;
    ...
}
```

We get to point B only by entering the loop, which means that the loop test must have evaluated to `true`. In other words, at point B it will always be true that `x` is not equal to `y`, so the assertion `x == y` will never be true at that point. But we don't know which of the two is larger. Therefore, the assertion `x > y` is sometimes true and sometimes false.

You might think that the assertion `z == 0` would always be true at point B because we were at point A just before we were at point B, but that is not the right answer. Remember that point B is inside a `while` loop. On the first iteration of the loop we will have been at point A just before reaching point B, but on later iterations of the loop we will have been inside the loop just before reaching point B. And if you look at the line of code just after point B, you will see that it increments `z`. There are no other modifications to the variable `z` inside the loop. Therefore, each time the body of the loop executes, `z` will increase by 1. So, `z` will be 0 at point B the first time through the loop, but it will be 1 on the second iteration, 2 on the third iteration, and so forth. Therefore, the right answer for the assertion `z == 0` at point B is that it is sometimes true, sometimes false. So, the second line of the table should look like this:

	<code>x > y</code>	<code>x == y</code>	<code>z == 0</code>
Point B	sometimes	never	sometimes

Point C is right after the increment of the variable `z`. There are no changes to the values of `x` and `y` between point B and point C, so the same answers apply at point C

for the assertions $x > y$ and $x == y$. The assertion $z == 0$ will never be true after the increment, even though z starts at 0 before the loop begins and there are no other manipulations of the variable inside the loop; once it is incremented, it will never be 0 again. Therefore, we can fill in the table for point C as follows:

	$x > y$	$x == y$	$z == 0$
Point C	sometimes	never	never

Points D and E are part of the `if/else` statement inside the `while` loop, so we can evaluate them as a pair. The `if/else` statement appears right after point C:

```
// Point C
if (x > y) {
    // Point D
    x = x / 10;
} else {
    // Point E
    y = y / 10;
}
```

No variables are changed between point C and points D and E. Java performs a test and branches in one of two directions. The `if/else` test determines whether x is greater than y . If the test is `true`, we go to point D. If not, we go to point E. So, for the assertion $x > y$, we know it is always true at point D and never true at point E. The assertion $x == y$ is a little more difficult to work out. We know it can never be true at point D, but could it be true at point E? Solely on the basis of the `if/else` test, the answer is yes. But remember that at point C the assertion could never be true. The values of x and y have not changed between point C and point E, so it still can never be true.

As for the assertion $z == 0$, the variable z hasn't changed between point C and points D and E, and z is not included in the test. So whatever we knew about z before still holds. Therefore, the right answers to fill in for points D and E are as follows:

	$x > y$	$x == y$	$z == 0$
Point D	always	never	never
Point E	never	never	never

Point F appears after the `if/else` statement. To determine the relationship between x and y at point F, we have to look at how the variables have changed. The `if/else` statement either divides x by 10 (if it is the larger value) or divides y by 10 (if it is the larger value). So, we have to ask whether it is possible for the assertion $x > y$ to be true at point F. The answer is yes. For example, x might have been 218 and y might have been 6 before the `if/else` statement. In that case, x would now be 21, which is still larger than y . But does it have to be larger than y ? Not necessarily. The

values might have been reversed, in which case y will be larger than x . So, that assertion is sometimes true and sometimes false at point F.

What about the assertion $x == y$? We know it doesn't have to be true because we have seen cases in which x is greater than y or y is greater than x . Is it possible for it to be true? Are there any values of x and y that would lead to this outcome? Consider the case in which x is 218 and y is 21. Then we would divide x by 10 to get 21, which would equal y . So, this assertion also is sometimes true and sometimes false.

There was no change to z between points D and E and point F, so we simply carry our answer down from the previous columns. So we would fill in the table as follows for point F:

	$x > y$	$x == y$	$z == 0$
Point F	sometimes	sometimes	never

Point G appears after the `while` loop:

```
while (x != y) {
    ...
}
// Point G
```

We can escape the `while` loop only if x becomes equal to y . So, at point G we know that the assertion $x == y$ is always true. That means that the assertion $x > y$ can never be true. The assertion $z == 0$ is a little tricky. At point F it was never true, so you might imagine that at point G it can never be true. But we weren't necessarily at point F just before we reached point G. We might never have entered the `while` loop at all, in which case we would have been at point A just before point G. At point A the variable z was equal to 0. Therefore, the right answer for this assertion is that it is sometimes true, sometimes false at point G. The final row of our table thus looks like this:

	$x > y$	$x == y$	$z == 0$
Point G	never	always	sometimes

When we combine this information, we can fill in our table as follows:

	$x > y$	$x == y$	$z == 0$
Point A	sometimes	sometimes	always
Point B	sometimes	never	sometimes
Point C	sometimes	never	never
Point D	always	never	never
Point E	never	never	never
Point F	sometimes	sometimes	never
Point G	never	always	sometimes

5.6 Case Study: NumberGuess

If we combine indefinite loops, the ability to check for user errors, and random number generation, it's possible for us to create guessing games in which the computer thinks of random numbers and the user tries to guess them. Let's consider an example game with the following rules. The computer thinks of a random two-digit number but keeps it secret from the player. We'll allow the program to accept positive numbers only, so the acceptable range of numbers is 00 through 99 inclusive. The player will try to guess the number the computer picked. If the player guesses correctly, the program will report the number of guesses that the player made.

To make the game more interesting, the computer will give the player a hint each time the user enters an incorrect guess. Specifically, the computer will tell the player how many digits from the guess are contained in the correct answer. The order of the digits doesn't affect the number of digits that match. For example, if the correct number is 57 and the player guesses 73, the computer will report one matching digit, because the correct answer contains a 7. If the player next guesses 75, the computer will report two matching digits. At this point the player knows that the computer's number must be 57, because 57 is the only two-digit number whose digits match those of 75.

Since the players will be doing a lot of console input, it's likely that they will type incorrect numbers or nonnumeric tokens by mistake. We'd like our guessing-game program to be robust against user input errors.

Initial Version without Hinting

In previous chapters, we've talked about the idea of iterative enhancement. Since this is a challenging program, we'll tackle it in stages. One of the hardest parts of the program is giving correct hints to the player. For now, we'll simply write a game that tells players whether they are correct or incorrect on each guess and, once the game is done, reports the number of guesses the players made. The program won't be robust against user input errors yet; that can be added later. To further simplify the game, rather than having the computer choose a random number, we'll choose a known value for the number so that the code can be tested more easily.

Since we don't know how many tries a player will need before correctly guessing the number, it seems that the main loop for this game will have to be a `while` loop. It might be tempting to write the code to match the following pseudocode:

```
// flawed number guess pseudocode
think of a number.
while (user has not guessed the number) {
    prompt and read a guess.
    report whether the guess was correct or incorrect.
}
```

But the problem with this pseudocode is that you can't start the `while` loop if you don't have a `guess` value from the player yet. The following code doesn't compile, because the variable `guess` isn't initialized when the loop begins:

```
// this code doesn't compile
int numGuesses = 0;
int number = 42; // computer always picks same number
int guess;

while (guess != number) {
    System.out.print("Your guess? ");
    guess = console.nextInt();
    numGuesses++;
    System.out.println("Incorrect.");
}

System.out.println("You got it right in " + numGuesses + " tries.");
```

It turns out that the game's main guess loop is a fencepost loop, because after each incorrect guess the program must print an "Incorrect" message (and later a hint). For n guesses, there are $n - 1$ hints. Recall the following general pseudocode for fencepost loops:

```
plant a post.
for (the length of the fence) {
    attach some wire.
    plant a post.
}
```

This particular problem is an indefinite fencepost using a `while` loop. Let's look at some more specific pseudocode. The "posts" are the prompts for guesses, and the "wires" are the "Incorrect" messages:

```
// specific number guess pseudocode
think of a number.
ask for the player's initial guess.
while (the guess is not the correct number) {
    inform the player that the guess was incorrect.
    ask for another guess.
}

report the number of guesses needed.
```


This pseudocode leads us to write the following Java program. Note that the computer always picks the value 42 in this version of the program:

```
1 import java.util.*;
2
3 public class NumberGuess1 {
4     public static void main(String[] args) {
5         Scanner console = new Scanner(System.in);
6         int number = 42; // always picks the same number
7
8         System.out.print("Your guess? ");
9         int guess = console.nextInt();
10        int numGuesses = 1;
11
12        while (guess != number) {
13            System.out.println("Incorrect.");
14            System.out.print("Your guess? ");
15            guess = console.nextInt();
16            numGuesses++;
17        }
18
19        System.out.println("You got it right in " +
20                           numGuesses + " tries.");
21    }
22 }
```

We can test our initial program to verify the code we've written so far. A sample dialogue looks like this:

```
Your guess? 65
Incorrect.
Your guess? 12
Incorrect.
Your guess? 34
Incorrect.
Your guess? 42
You got it right in 4 tries.
```

Randomized Version with Hinting

Now that we've tested the code to make sure our main game loops, let's make the game random by choosing a random value between 00 and 99 inclusive. To do so, we'll create a `Random` object and call its `nextInt` method, specifying the maximum

value. Remember that the value passed to `nextInt` should be one more than the desired maximum, so we'll pass 100:

```
// pick a random number between 00 and 99 inclusive
Random rand = new Random();
int number = rand.nextInt(100);
```

The next important feature our game should have is to give a hint when the player makes an incorrect guess. The tricky part is figuring out how many digits of the player's guess match the correct number. Since this code is nontrivial to write, let's make a method called `matches` that does the work for us. To figure out how many digits match, the `matches` method needs to use the guess and the correct number as parameters. It will return the number of matching digits. Therefore, its header should look like this:

```
public static int matches(int number, int guess) {
    ...
}
```

Our algorithm must count the number of matching digits. Either digit from the guess can match either digit from the correct number. Since the digits are somewhat independent—that is, whether the ones digit of the guess matches is independent of whether the tens digit matches—we should use sequential `if` statements rather than an `if/else` statement to represent these conditions.

The digit-matching algorithm has one special case. If the player guesses a number such as 33 that contains two of the same digit, and if that digit is contained in the correct answer (say the correct answer is 37), it would be misleading to report that two digits match. It makes more sense for the program to report one matching digit. To handle this case, our algorithm must check whether the guess contains two of the same digit and consider the second digit of the guess to be a match only if it is different from the first.

Here is the pseudocode for the algorithm:

```
matches = 0.
if (the first digit of the guess matches
    either digit of the correct number) {
    we have found one match.
}

if (the second digit of the guess is different from the first digit,
    AND it matches either digit of the correct number) {
    we have found another match.
}
```

We need to be able to split the correct number and the guess into the two digits that compose each so that we can compare them. Recall from the Boolean Zen section that we can use the division and remainder operators to express the digits of any two-digit number n as $n / 10$ for the tens digit and $n \% 10$ for the ones digit.

Let's write the statement that compares the tens digit of the guess against the correct answer. Since the tens digit of the guess can match either of the correct number's digits, we'll use an OR test with the `||` operator:

```
int matches = 0;

// check the first digit for a match
if (guess / 10 == number / 10 || guess / 10 == number % 10) {
    matches++;
}
```

Writing the statement that compares the ones digit of the guess against the correct answer is slightly trickier, because we have to take into consideration the special case described previously (in which both digits of the guess are the same). We'll account for this by counting the second digit as a match only if it is unique *and* matches a digit from the correct number:

```
// check the second digit for a match
if (guess / 10 != guess % 10 &&
    (guess % 10 == number / 10 || guess % 10 == number % 10)) {
    matches++;
}
```

The following version of the program uses the hinting code we've just written. It also adds the randomly chosen number and a brief introduction to the program:

```
1 // Two-digit number-guessing game with hinting.
2 import java.util.*;
3
4 public class NumberGuess2 {
5     public static void main(String[] args) {
6         System.out.println("Try to guess my two-digit");
7         System.out.println("number, and I'll tell you how");
8         System.out.println("many digits from your guess");
9         System.out.println("appear in my number.");
10        System.out.println();
11
12        Scanner console = new Scanner(System.in);
13
14        // pick a random number from 0 to 99 inclusive
```

```
15     Random rand = new Random();
16     int number = rand.nextInt(100);
17
18     // get first guess
19     System.out.print("Your guess? ");
20     int guess = console.nextInt();
21     int numGuesses = 1;
22
23     // give hints until correct guess is reached
24     while (guess != number) {
25         int numMatches = matches(number, guess);
26         System.out.println("Incorrect (hint: " +
27                             numMatches + " digits match)");
28         System.out.print("Your guess? ");
29         guess = console.nextInt();
30         numGuesses++;
31     }
32
33     System.out.println("You got it right in " +
34                         numGuesses + " tries.");
35 }
36
37 // returns how many digits from the given
38 // guess match digits from the given correct number
39 public static int matches(int number, int guess) {
40     int numMatches = 0;
41
42     if (guess / 10 == number / 10 ||
43         guess / 10 == number % 10) {
44         numMatches++;
45     }
46
47     if (guess / 10 != guess % 10 &&
48         (guess % 10 == number / 10 ||
49          guess % 10 == number % 10)) {
50         numMatches++;
51     }
52
53     return numMatches;
54 }
55 }
```

The following is a sample log of the program execution:

```
Try to guess my two-digit
number, and I'll tell you how
many digits from your guess
appear in my number.

Your guess? 13
Incorrect (hint: 0 digits match)
Your guess? 26
Incorrect (hint: 0 digits match)
Your guess? 78
Incorrect (hint: 1 digits match)
Your guess? 79
Incorrect (hint: 1 digits match)
Your guess? 70
Incorrect (hint: 2 digits match)
Your guess? 7
You got it right in 6 tries.
```

Final Robust Version

The last major change we'll make to our program is to make it robust against invalid user input. There are two types of bad input that we may see:

1. Nonnumeric tokens.
2. Numbers outside the range of 0–99.

Let's deal with these cases one at a time. Recall the `getInt` method that was discussed earlier in this chapter. It repeatedly prompts the user for input until an integer is typed. Here is its header:

```
public static int getInt(Scanner console, String prompt)
```

We can make use of `getInt` to get an integer between 0 and 99. We'll repeatedly call `getInt` until the integer that is returned is within the acceptable range. The postcondition we require before we can stop prompting for guesses is:

```
guess >= 0 && guess <= 99
```

To ensure that this postcondition is met, we can use a `while` loop that tests for the opposite condition. Using De Morgan's law, we know that the opposite of the previous test would be the following:

```
guess < 0 || guess > 99
```

The reversed test is used in our new `getGuess` method to get a valid guess between 0 and 99. Now whenever we want to read user input in the main program, we'll call `getGuess`. It's useful to separate the input prompting in this way, to make sure that we don't accidentally count invalid inputs as guesses.

The final version of our code is the following:

```

1 // Robust two-digit number-guessing game with hinting.
2 import java.util.*;
3
4 public class NumberGuess3 {
5     public static void main(String[] args) {
6         giveIntro();
7         Scanner console = new Scanner(System.in);
8
9         // pick a random number from 0 to 99 inclusive
10        Random rand = new Random();
11        int number = rand.nextInt(100);
12
13        // get first guess
14        int guess = getGuess(console);
15        int numGuesses = 1;
16
17        // give hints until correct guess is reached
18        while (guess != number) {
19            int numMatches = matches(number, guess);
20            System.out.println("Incorrect (hint: " +
21                               numMatches + " digits match)");
22            guess = getGuess(console);
23            numGuesses++;
24        }
25
26        System.out.println("You got it right in " +
27                           numGuesses + " tries.");
28    }
29
30    public static void giveIntro() {
31        System.out.println("Try to guess my two-digit");
32        System.out.println("number, and I'll tell you how");
33        System.out.println("many digits from your guess");
34        System.out.println("appear in my number.");
35        System.out.println();
36    }
37
38    // returns # of matching digits between the two numbers

```



```
39     // pre: number and guess are unique two-digit numbers
40     public static int matches(int number, int guess) {
41         int numMatches = 0;
42
43         if (guess / 10 == number / 10 ||
44             guess / 10 == number % 10) {
45             numMatches++;
46         }
47
48         if (guess / 10 != guess % 10 &&
49             (guess % 10 == number / 10 ||
50              guess % 10 == number % 10)) {
51             numMatches++;
52         }
53
54         return numMatches;
55     }
56
57     // prompts until a number in proper range is entered
58     // post: guess is between 0 and 99
59     public static int getGuess(Scanner console) {
60         int guess = getInt(console, "Your guess? ");
61         while (guess < 0 || guess >= 100) {
62             System.out.println("Out of range; try again.");
63             guess = getInt(console, "Your guess? ");
64         }
65
66         return guess;
67     }
68
69     // prompts until a valid number is entered
70     public static int getInt(Scanner console, String prompt) {
71         System.out.print(prompt);
72         while (!console.hasNextInt()) {
73             console.next(); // to discard the input
74             System.out.println("Not an integer; try again.");
75             System.out.print(prompt);
76         }
77         return console.nextInt();
78     }
79 }
```

The following sample log of execution demonstrates the new input robustness of this program:

```
Try to guess my two-digit
number, and I'll tell you how
many digits from your guess
appear in my number.

Your guess? 12
Incorrect (hint: 0 digits match)
Your guess? okay
Not an integer; try again.
Your guess? 34
Incorrect (hint: 1 digits match)
Your guess? 35
Incorrect (hint: 1 digits match)
Your guess? 67
Incorrect (hint: 0 digits match)
Your guess? 89
Incorrect (hint: 0 digits match)
Your guess? 3
Incorrect (hint: 2 digits match)
Your guess? 300
Out of range; try again.
Your guess? 30
You got it right in 7 tries.
```

Notice that we're careful to comment our code to document relevant preconditions and postconditions of our methods. The precondition of the `matches` method is that the two parameters are unique two-digit numbers. The postcondition of our new `getGuesses` method is that it returns a guess between 0 and 99 inclusive. Also, note that the program does not count invalid input (`okay` and `300` in the previous sample log of execution) as guesses.

Chapter Summary

Java has a `while` loop in addition to its `for` loop. The `while` loop can be used to write indefinite loops that keep executing until some condition fails.

Priming a loop means setting the values of variables that will be used in the loop test, so that the test will be sure to succeed the first time and the loop will execute.

Java can generate pseudorandom numbers using objects of the `Random` class.

The `do/while` loop is a variation on the `while` loop that performs its loop test at the end of the loop body. A `do/while` loop is guaranteed to execute its body at least once.

A fencepost loop executes a “loop-and-a-half” by executing part of a loop’s body once before the loop begins.

A sentinel loop is a kind of fencepost loop that repeatedly processes input until it is passed a particular value, but does not process the special value.

The `boolean` primitive type represents logical values of either `true` or `false`. Boolean expressions are used as tests in `if` statements and loops. Boolean expressions can use relational operators such as `<` or `!=` as well as logical operators such as `&&` or `!`.

Complex Boolean tests with logical operators such as `&&` or `||` are evaluated lazily: If the overall result is clear from evaluating the first part of the expression, later parts are not evaluated. This is called short-circuited evaluation.

Boolean variables (sometimes called “flags”) can store Boolean values and can be used as loop tests.

A complex Boolean expression can be negated using a set of rules known as De Morgan’s laws, in which each sub-expression is negated and all AND and OR operations are swapped.

A robust program checks for errors in user input. Better robustness can be achieved by looping and reprompting the user to enter input when he or she types bad input. The `Scanner` class has methods like `hasNextInt` that you can use to “look ahead” for valid input.

Assertions are logical statements about a particular point in a program. Assertions are useful for proving properties about how a program will execute. Two useful types of assertions are preconditions and postconditions, which are claims about what will be true before and after a method executes.

Self-Check Problems

Section 5.1: The while Loop

- For each of the following while loops, state how many times the loop will execute its body. Remember that “zero,” “infinity,” and “unknown” are legal answers. Also, what is the output of the code in each case?

```
a. int x = 1;
   while (x < 100) {
       System.out.print(x + " ");
       x += 10;
   }

b. int max = 10;
   while (max < 10) {
       System.out.println("count down: " + max);
       max--;
   }

c. int x = 250;
   while (x % 3 != 0) {
       System.out.println(x);
   }
```

```

d. int x = 2;
   while (x < 200) {
       System.out.print(x + " ");
       x *= x;
   }
e. String word = "a";
   while (word.length() < 10) {
       word = "b" + word + "b";
   }
   System.out.println(word);
f. int x = 100;
   while (x > 0) {
       System.out.println(x / 10);
       x = x / 2;
   }

```

2. Convert each of the following for loops into an equivalent while loop:

```

a. for (int n = 1; n <= max; n++) {
    System.out.println(n);
}
b. int total = 25;
   for (int number = 1; number <= (total / 2); number++) {
       total = total - number;
       System.out.println(total + " " + number);
   }
c. for (int i = 1; i <= 2; i++) {
    for (int j = 1; j <= 3; j++) {
        for (int k = 1; k <= 4; k++) {
            System.out.print("*");
        }
        System.out.print("!");
    }
    System.out.println();
}
d. int number = 4;
   for (int count = 1; count <= number; count++) {
       System.out.println(number);
       number = number / 2;
   }

```

3. Consider the following method:

```

public static void mystery(int x) {
    int y = 1;
    int z = 0;
    while (2 * y <= x) {
        y = y * 2;
    }
}

```



```

        z++;
    }
    System.out.println(y + " " + z);
}

```

For each of the following calls, indicate the output that the preceding method produces:

```

mystery(1);
mystery(6);
mystery(19);
mystery(39);
mystery(74);

```

4. Consider the following method:

```

public static void mystery(int x) {
    int y = 0;
    while (x % 2 == 0) {
        y++;
        x = x / 2;
    }
    System.out.println(x + " " + y);
}

```

For each of the following calls, indicate the output that the preceding method produces:

```

mystery(19);
mystery(42);
mystery(48);
mystery(40);
mystery(64);

```

5. Consider the following code:

```

Random rand = new Random();
int a = rand.nextInt(100);
int b = rand.nextInt(20) + 50;
int c = rand.nextInt(20 + 50);
int d = rand.nextInt(100) - 20;
int e = rand.nextInt(10) * 4;

```

What range of values can each variable (a, b, c, d, and e) have?

6. Write code that generates a random integer between 0 and 10 inclusive.
7. Write code that generates a random odd integer (not divisible by 2) between 50 and 99 inclusive.
8. For each of the do/while loops that follow, state the number of times that the loop will execute its body. Remember that "zero," "infinity," and "unknown" are legal answers. Also, what is the output of the code in each case?

```

a. int x = 1;
   do {

```



```

        System.out.print(x + " ");
        x = x + 10;
    } while (x < 100);
b. int max = 10;
    do {
        System.out.println("count down: " + max);
        max--;
    } while (max < 10);
c. int x = 250;
    do {
        System.out.println(x);
    } while (x % 3 != 0);
d. int x = 100;
    do {
        System.out.println(x);
        x = x / 2;
    } while (x % 2 == 0);
e. int x = 2;
    do {
        System.out.print(x + " ");
        x *= x;
    } while (x < 200);
f. String word = "a";
    do {
        word = "b" + word + "b";
    } while (word.length() < 10);
    System.out.println(word);
g. int x = 100;
    do {
        System.out.println(x / 10);
        x = x / 2;
    } while (x > 0);
h. String str = "/\\\\";
    do {
        str += str;
    } while (str.length() < 10);
    System.out.println(str);

```

9. Write a do/while loop that repeatedly prints a certain message until the user tells the program to stop. The do/while is appropriate because the message should always be printed at least one time, even if the user types n after the first message appears. The message to be printed is as follows:

```

She sells seashells by the seashore.
Do you want to hear it again? y
She sells seashells by the seashore.
Do you want to hear it again? y

```

She sells seashells by the seashore.

Do you want to hear it again? **n**

10. Write a method called `zeroDigits` that accepts an integer parameter and returns the number of digits in the number that have the value 0. For example, the call `zeroDigits(5024036)` should return 2, and `zeroDigits(743)` should return 0. The call `zeroDigits(0)` should return 1. (We suggest you use a `do/while` loop in your solution.)
11. Write a `do/while` loop that repeatedly prints random numbers between 0 and 1000 until a number above 900 is printed. At least one line of output should always be printed, even if the first random number is above 900. Here is a sample execution:

Random number: 235

Random number: 15

Random number: 810

Random number: 147

Random number: 915

Section 5.2: Fencepost Algorithms

12. Consider the flawed method `printLetters` that follows, which accepts a `String` as its parameter and attempts to print the letters of the `String`, separated by dashes. For example, the call of `printLetters("Rabbit")` should print `R-a-b-b-i-t`. The following code is incorrect:

```
public static void printLetters(String text) {
    for (int i = 0; i < text.length(); i++) {
        System.out.print(text.charAt(i) + "-");
    }
    System.out.println(); // to end the line of output
}
```

What is wrong with the code? How can it be corrected to produce the desired behavior?

13. Write a sentinel loop that repeatedly prompts the user to enter a number and, once the number `-1` is typed, displays the maximum and minimum numbers that the user entered. Here is a sample dialogue:

```
Type a number (or -1 to stop): 5
Type a number (or -1 to stop): 2
Type a number (or -1 to stop): 17
Type a number (or -1 to stop): 8
Type a number (or -1 to stop): -1
Maximum was 17
Minimum was 2
```

If `-1` is the first number typed, no maximum or minimum should be printed. In this case, the dialogue would look like this:

```
Type a number (or -1 to stop): -1
```

Section 5.3: The boolean Type

14. Consider the following variable declarations:

```
int x = 27;
int y = -1;
```

```
int z = 32;
boolean b = false;
```

What is the value of each of the following Boolean expressions?

- a. !b
- b. b || true
- c. (x > y) && (y > z)
- d. (x == y) || (x <= z)
- e. !(x % 2 == 0)
- f. (x % 2 != 0) && b
- g. b && !b
- h. b || !b
- i. (x < y) == b
- j. !(x / 2 == 13) || b || (z * 3 == 96)
- k. (z < x) == false
- l. !((x > 0) && (y < 0))

15. Write a method called `isVowel` that accepts a character as input and returns `true` if that character is a vowel (a, e, i, o, or u). For an extra challenge, make your method case-insensitive.
16. The following code attempts to examine a number and return whether that number is prime (i.e., has no factors other than 1 and itself). A flag named `prime` is used. However, the Boolean logic is not implemented correctly, so the method does not always return the correct answer. In what cases does the method report an incorrect answer? How can the code be changed so that it will always return a correct result?

```
public static boolean isPrime(int n) {
    boolean prime = true;
    for (int i = 2; i < n; i++) {
        if (n % i == 0) {
            prime = false;
        } else {
            prime = true;
        }
    }

    return prime;
}
```

17. The following code attempts to examine a `String` and return whether it contains a given letter. A flag named `found` is used. However, the Boolean logic is not implemented correctly, so the method does not always return the correct answer. In what cases does the method report an incorrect answer? How can the code be changed so that it will always return a correct result?

```
public static boolean contains(String str, char ch) {
    boolean found = false;
    for (int i = 0; i < str.length(); i++) {
        if (str.charAt(i) == ch) {
            found = true;
        }
    }
}
```

```

    } else {
        found = false;
    }
}
return found;
}

```

18. Using "Boolean Zen," write an improved version of the following method, which returns whether the given String starts and ends with the same character:

```

public static boolean startEndSame(String str) {
    if (str.charAt(0) == str.charAt(str.length() - 1)) {
        return true;
    } else {
        return false
    }
}

```

19. Using "Boolean Zen," write an improved version of the following method, which returns whether the given number of cents would require any pennies (as opposed to being an amount that could be made exactly using coins other than pennies):

```

public static boolean hasPennies(int cents) {
    boolean nickelsOnly = (cents % 5 == 0);
    if (nickelsOnly == true) {
        return false;
    } else {
        return true;
    }
}

```

20. Consider the following method:

```

public static int mystery(int x, int y) {
    while (x != 0 && y != 0) {
        if (x < y) {
            y -= x;
        } else {
            x -= y;
        }
    }
    return x + y;
}

```

For each of the following calls, indicate the value that is returned:

```

mystery(3, 3)
mystery(5, 3)
mystery(2, 6)

```



```
mystery(12, 18)
mystery(30, 75)
```

21. The following code is a slightly modified version of actual code that was in the Microsoft Zune music player in 2008. The code attempts to calculate today's date by determining how many years and days have passed since 1980. Assume the existence of methods for getting the total number of days since 1980 and for determining whether a given year is a leap year:

```
int days = getTotalDaysSince1980();
year = 1980;
while (days > 365) { // subtract out years
    if (isLeapYear(year)) {
        if (days > 366) {
            days -= 366;
            year += 1;
        }
    } else {
        days -= 365;
        year += 1;
    }
}
```

Thousands of Zune players locked up on January 1, 2009, the first day after the end of a leap year since the Zune was released. (Microsoft quickly released a patch to fix the problem.) What is the problem with the preceding code, and in what cases will it exhibit incorrect behavior? How can it be fixed?

22. Consider the following variable declarations:

```
int x = 27;
int y = -1;
int z = 32;
boolean b = false;
```

Write a new Boolean expression that is the negation of each of the following Boolean expressions. Use De Morgan's laws rather than simply writing a ! at the beginning of each entire expression.

- b
- (x > y) && (y > z)
- (x == y) || (x <= z)
- (x % 2 != 0) && b
- (x / 2 == 13) || b || (2 * 3 == 96)
- (z < x) && (z > y) || x >= y

Section 5.4: User Errors

23. The following code is not robust against invalid user input. Describe how to change the code so that it will not proceed until the user has entered a valid age and grade point average (GPA). Assume that any int is a legal age and that any double is a legal GPA.

```
Scanner console = new Scanner(System.in);
System.out.print("Type your age: ");
int age = console.nextInt();
```



```
System.out.print("Type your GPA: ");
double gpa = console.nextDouble();
```

For an added challenge, modify the code so that it rejects invalid ages (for example, numbers less than 0) and GPAs (say, numbers less than 0.0 or greater than 4.0).

24. Consider the following code:

```
Scanner console = new Scanner(System.in);
System.out.print("Type something for me! ");

if (console.hasNextInt()) {
    int number = console.nextInt();
    System.out.println("Your IQ is " + number);
} else if (console.hasNext()) {
    String token = console.next();
    System.out.println("Your name is " + token);
}
```

What is the output when the user types the following values?

- a. Jane
- b. 56
- c. 56.2

25. Write a piece of code that prompts the user for a number and then prints a different message depending on whether the number was an integer or a real number. Here are two sample dialogues:

```
Type a number: 42.5
You typed the real number 42.5
```

```
Type a number: 3
You typed the integer 3
```

26. Write code that prompts for three integers, averages them, and prints the average. Make your code robust against invalid input. (You may want to use the `getInt` method discussed in this chapter.)

Section 5.5: Assertions and Program Logic

27. Identify the various assertions in the following code as being always true, never true, or sometimes true and sometimes false at various points in program execution. The comments indicate the points of interest:

```
public static int mystery(Scanner console, int x) {
    int y = console.nextInt();
    int count = 0;

    // Point A
    while (y < x) {
        // Point B
        if (y == 0) {
```

```

        count++;
        // Point C
    }
    y = console.nextInt();
    // Point D
}

// Point E
return count;
}

```

Categorize each assertion at each point with ALWAYS, NEVER, or SOMETIMES:

	<code>y < x</code>	<code>y == 0</code>	<code>count > 0</code>
Point A			
Point B			
Point C			
Point D			
Point E			

28. Identify the various assertions in the following code as being always true, never true, or sometimes true and sometimes false at various points in program execution. The comments indicate the points of interest:

```

public static int mystery(int n) {
    Random r = new Random();
    int a = r.nextInt(3) + 1;
    int b = 2;
    // Point A
    while (n > b) {
        // Point B
        b = b + a;
        if (a > 1) {
            n--;
            // Point C
            a = r.nextInt(b) + 1;
        } else {
            a = b + 1;

            // Point D
        }
    }
    // Point E
    return n;
}

```

Categorize each assertion at each point with ALWAYS, NEVER, or SOMETIMES:

	$n > b$	$a > 1$	$b > a$
Point A			
Point B			
Point C			
Point D			
Point E			

29. Identify the various assertions in the following code as being always true, never true, or sometimes true and sometimes false at various points in program execution. The comments indicate the points of interest:

```
public static int mystery(Scanner console) {
    int prev = 0;
    int count = 0;
    int next = console.nextInt();
    // Point A
    while (next != 0) {
        // Point B
        if (next == prev) {
            // Point C
            count++;
        }
        prev = next;
        next = console.nextInt();
        // Point D
    }
    // Point E
    return count;
}
```

Categorize each assertion at each point with ALWAYS, NEVER, or SOMETIMES:

	$next == 0$	$prev == 0$	$next == prev$
Point A			
Point B			
Point C			
Point D			
Point E			

Exercises

1. Write a method called `showTwos` that shows the factors of 2 in a given integer. For example, consider the following calls:

```
showTwos(7);
showTwos(18);
showTwos(68);
showTwos(120);
```

These calls should produce the following output:

```
7 = 7
18 = 2 * 9
68 = 2 * 2 * 17
120 = 2 * 2 * 2 * 15
```

2. Write a method called `gcd` that accepts two integers as parameters and returns the greatest common divisor (GCD) of the two numbers. The GCD of two integers *a* and *b* is the largest integer that is a factor of both *a* and *b*.

One efficient way to compute the GCD of two numbers is to use Euclid's algorithm, which states the following:

$\text{GCD}(a, b) = \text{GCD}(b, a \% b)$

$\text{GCD}(a, 0) = \text{Absolute value of } a$

3. Write a method called `toBinary` that accepts an integer as a parameter and returns a `String` containing that integer's binary representation. For example, the call of `printBinary(44)` should return "101100".
4. Write a method called `randomX` that prints a lines that contain a random number of "x" characters (between 5 and 20 inclusive) until it prints a line that contains 16 or more characters. For example, the output might look like the following:

```
xxxxxxx
xxxxxxxxxxxxxxxx
xxxxxxxxxxxxx
xxxxxxxxxxxxxxxx
xxxxxx
xxxxxxxxxxxx
xxxxxxxxxxxxxxxx
```

5. Write a method called `randomLines` that prints between 5 and 10 random strings of letters (between "a" and "z"), one per line. Each string should have random length of up to 80 characters.
6. Write a method called `makeGuesses` that guesses numbers between 1 and 50 inclusive until it makes a guess of at least 48. It should report each guess and at the end should report the total number of guesses made. Here is a sample execution:

```
guess = 43
guess = 47
guess = 45
guess = 27
guess = 49
total guesses = 5
```


7. Write a method called `diceSum` that accepts a `Scanner` for the console as a parameter and prompts for a desired sum, then repeatedly simulates the rolling of 2 six-sided dice until their sum is the desired sum. Here is a sample dialogue with the user:

Desired dice sum: 9

4 and 3 = 7

3 and 5 = 8

5 and 6 = 11

5 and 6 = 11

1 and 5 = 6

6 and 3 = 9

8. Write a method called `randomWalk` that performs steps of a random one-dimensional walk. The random walk should begin at position 0. On each step, you should either increase or decrease the position by 1 (each with equal probability). Your code should continue making steps until a position of 3 or -3 is reached, and then report the maximum position that was reached during the walk. The output should look like the following:

position = 1

position = 0

position = -1

position = -2

position = -1

position = -2

position = -3

max position = 1

9. Write a method called `printFactors` that accepts an integer as its parameter and uses a fencepost loop to print the factors of that number, separated by the word "and". For example, the factors of the number 24 should print as the following:

1 and 2 and 3 and 4 and 6 and 8 and 12 and 24

You may assume that the parameter's value is greater than 0, or you may throw an exception if it is 0 or negative. Your method should print nothing if the empty string (" ") is passed.

10. Write a method called `hopscotch` that accepts an integer number of "hops" as its parameter and prints a pattern of numbers that resembles a hopscotch board. A "hop" is a three-number sequence where the output shows two numbers on a line, followed by one number on its own line. 0 hops is a board up to 1; one hop is a board up to 4; two hops is a board up to 7; and so on. For example, the call of `hopscotch(3);` should print the following output:

1

2 3

4

5 6

7

8 9

10

A call of `hopscotch(0);` should print only the number 1. If it is passed a negative value, the method should produce no output.

11. Write a method called `threeHeads` that repeatedly flips a coin until the results of the coin toss are three heads in a row. You should use a `Random` object to make it equally likely that a head or a tail will appear. Each time the coin is flipped, display H for heads or T for tails. When three heads in a row are flipped, the method should print a congratulatory message. Here is a possible output of a call to the method:

```
T T H T T T H T H T H H H
Three heads in a row!
```

12. Write a method called `printAverage` that uses a sentinel loop to repeatedly prompt the user for numbers. Once the user types any number less than zero, the method should display the average of all nonnegative numbers typed. Display the average as a `double`. Here is a sample dialogue with the user:

```
Type a number: 7
Type a number: 4
Type a number: 16
Type a number: -4
Average was 9.0
```

If the first number that the user types is negative, do not print an average:

```
Type a number: -2
```

13. Write a method called `consecutive` that accepts three integers as parameters and returns `true` if they are three consecutive numbers—that is, if the numbers can be arranged into an order such that, assuming some integer k , the parameters' values are k , $k + 1$, and $k + 2$. Your method should return `false` if the integers are not consecutive. Note that order is not significant; your method should return the same result for the same three integers passed in any order.

For example, the calls `consecutive(1, 2, 3)`, `consecutive(3, 2, 4)`, and `consecutive(-10, -8, -9)` would return `true`. The calls `consecutive(3, 5, 7)`, `consecutive(1, 2, 2)`, and `consecutive(7, 7, 9)` would return `false`.

14. Write a method called `hasMidpoint` that accepts three integers as parameters and returns `true` if one of the integers is the midpoint between the other two integers; that is, if one integer is exactly halfway between them. Your method should return `false` if no such midpoint relationship exists. For example, the call `hasMidpoint(7, 4, 10)` should return `true` because 7 is halfway between 4 and 10. By contrast, the call `hasMidpoint(9, 15, 8)` should return `false` because no integer is halfway between the other two. The integers could be passed in any order; the midpoint could be the 1st, 2nd, or 3rd. You must check all cases. If your method is passed three of the same value, return `true`.

15. Write a method called `dominant` that accepts three integers as parameters and returns `true` if any one of the three integers is larger than the sum of the other two integers. The integers might be passed in any order, so the largest value could be any of the three. For example, the call `dominant(4, 9, 2)` returns `true` because 9 is larger than $4 + 2$. Assume that none of the numbers is negative.

16. Write a method called `anglePairs` that accepts three angles (integers), measured in degrees, as parameters and returns whether or not there exist both complementary and supplementary angles among the three angles passed. Two angles are complementary if their sum is exactly 90 degrees; two angles are supplementary if their sum is exactly 180 degrees. Therefore, the method should return `true` if any two of the three angles add up to 90 degrees and also any two of the three angles add up to 180 degrees. For example, the call `anglePairs(120, 60, 30)` returns `true`. Assume that each angle passed is nonnegative.

17. Write a method called `monthApart` that accepts four integer parameters, $m1$, $d1$, $m2$, and $d2$, representing two calendar dates. Each date consists of a month (1 through 12) and a day (1 through the number of days in that month

- [28–31]). Assume that all parameter values passed are valid. The method should return `true` if the dates are at least a month apart and `false` otherwise. For example, the call of `monthApart(4, 15, 5, 22)` would return `true` while the call of `monthApart(9, 19, 10, 17)` would return `false`. Assume that all the dates in this problem occur during the same year. Note that the first date could come before or after the second date.
18. Write a method called `digitSum` that accepts an integer as a parameter and returns the sum of the digits of that number. For example, the call `digitSum(29107)` returns $2 + 9 + 1 + 0 + 7$ or 19. For negative numbers, return the same value that would result if the number were positive. For example, `digitSum(-456)` returns $4 + 5 + 6$ or 15. The call `digitSum(0)` returns 0.
 19. Write a method called `firstDigit` that returns the first (most significant) digit of an integer. For example, `firstDigit(3572)` should return 3. It should work for negative numbers as well; `firstDigit(-947)` should return 9.
 20. Write a method called `digitRange` that accepts an integer as a parameter and returns the range of values of its digits. The range is defined as 1 more than the difference between the largest and smallest digit value. For example, the call of `digitRange(68437)` would return 6 because the largest digit value is 8 and the smallest is 3, so $8 - 3 + 1 = 6$. If the number contains only one digit, return 1. You should solve this problem without using a `String`.
 21. Write a method called `swapDigitPairs` that accepts an integer n as a parameter and returns a new integer whose value is similar to n 's but with each pair of digits swapped in order. For example, the call of `swapDigitPairs(482596)` would return 845269. Notice that the 9 and 6 are swapped, as are the 2 and 5, and the 4 and 8. If the number contains an odd number of digits, leave the leftmost digit in its original place. For example, the call of `swapDigitPairs(1234567)` would return 1325476. You should solve this problem without using a `String`.
 22. Write a method called `allDigitsOdd` that returns whether every digit of a positive integer is odd. Return `true` if the number consists entirely of odd digits (1, 3, 5, 7, 9) and `false` if any of its digits are even (0, 2, 4, 6, 8). For example, the call `allDigitsOdd(135319)` returns `true` but `allDigitsOdd(9145293)` returns `false`.
 23. Write a method called `hasAnOddDigit` that returns whether a given positive integer has at least one digit whose value is odd. Return `true` if the number has at least one odd digit and `false` if none of its digits are odd. For example, the call `hasAnOddDigit(4822116)` should return `true` and `hasAnOddDigit(2448)` should return `false`.
 24. Write a method called `isAllVowels` that returns whether a string consists entirely of vowels (a, e, i, o, or u, case-insensitively). If and only if every character of the string is a vowel, your method should return `true`. For example, the call `isAllVowels("eIEiO")` returns `true` and `isAllVowels("oink")` returns `false`. You should return `true` if passed the empty string, since it does not contain any non-vowel characters.

Programming Projects

1. Write an interactive program that reads lines of input from the user and converts each line into "Pig Latin." Pig Latin is English with the initial consonant sound moved to the end of each word, followed by "ay." Words that begin with vowels simply have an "ay" appended. For example, the phrase

The deepest shade of mushroom blue

would have the following appearance in Pig Latin:

e-Thay eepest-day ade-shay of-ay ushroom-may ue-blay

Terminate the program when the user types a blank line.

2. Write a reverse Hangman game in which the user thinks of a word and the computer tries to guess the letters in that word. The user tells the computer how many letters the word contains.
3. Write a program that plays a guessing game with the user. The program should generate a random number between 1 and some maximum (such as 100), then prompt the user repeatedly to guess the number. When the user guesses incorrectly, the game should give the user a hint about whether the correct answer is higher or lower than the guess. Once the user guesses correctly, the program should print a message showing the number of guesses that the user made.

Consider extending this program by making it play multiple games until the user chooses to stop and then printing statistics about the player's total and average number of guesses.

4. Write a program that plays a reverse guessing game with the user. The user thinks of a number between 1 and 10, and the computer repeatedly tries to guess it by guessing random numbers. It's fine for the computer to guess the same random number more than once. At the end of the game, the program reports how many guesses it made. Here is a sample execution:

```
This program has you, the user, choose a number
between 1 and 10. Then I, the computer, will try
my best to guess it.
Is it 8? (y/n) n
Is it 7? (y/n) n
Is it 5? (y/n) n
Is it 1? (y/n) n
Is it 8? (y/n) n
Is it 1? (y/n) n
Is it 9? (y/n) y
I got your number of 9 correct in 7 guesses.
```

For an added challenge, consider having the user hint to the computer whether the correct number is higher or lower than the computer's guess. The computer should adjust its range of random guesses on the basis of the hint.

5. Write a game that plays many rounds of Rock Paper Scissors. The user and computer will each choose between three items: rock (defeats scissors, but loses to paper), paper (defeats rock, but loses to scissors), and scissors (defeats paper, but loses to rock). If the player and computer choose the same item, the game is a tie.

You could extend this program to include different algorithmic strategies for choosing the best item. Should the computer pick randomly? Should it always pick a particular item or a repeating pattern of items? Should it count the number of times the opponent chooses various items and base its strategy on this history?

6. Write a program that draws a graphical display of a 2D random walk using a `DrawingPanel`. Start a pixel walker in the middle of the panel. On each step, choose to move 1 pixel up, down, left, or right, then redraw the pixel. (You can draw a single pixel by drawing a rectangle of size 1×1 .)
7. Write a program that plays the dice game "Pig." Pig is a two-player game where the players take turns repeatedly rolling a single 6-sided die; a player repeatedly rolls the die until one of two events occurs. Either the player chooses to stop rolling, in which case the sum of that player's rolls are added to his/her total points; or if the player rolls a 1 at any time, all points for that turn are lost and the turn ends immediately. The first player to reach a score of at least 100 points wins.