Chapter 6 Iteration

CHAPTER GOALS

- To be able to program loops with the `while`, , and `for` statements
- To avoid infinite loops and off-by-one errors
- To understand nested loops
- To learn how to process input
- To implement simulations

T To learn about the debugger

This chapter presents the various iteration constructs of the Java language. These constructs execute one or more statements repeatedly until a goal is reached. You will see how the techniques that you learn in this chapter can be applied to the processing of input data and the programming of simulations.

6.1 While Loops

In this chapter you will learn how to write programs that repeatedly execute one or more statements. We will illustrate these concepts by looking at typical investment situations. Consider a bank account with an initial balance of $10,000 that earns 5% interest. The interest is computed at the end of every year on the current balance and then deposited into the bank account. For example, after the first year, the account has earned $500 (5% of $10,000) of interest. The interest gets added to the bank account. Next year, the interest is $525 (5% of $10,500), and the balance is $11,025. Table 1 shows how the balance grows in the first five years.

How many years does it take for the balance to reach $20,000? Of course, it won't take longer than 20 years, because at least $500 is added to the bank account each year. But it will take less than 20 years, because interest is computed on increasingly larger balances. To know the exact answer, we will write a program that repeatedly adds interest until the balance is reached.

In Java, the `while` statement implements such a repetition. The construct

A `while` statement executes a block of code repeatedly. A condition controls how often the loop is executed.

```
while (condition
  statement
```

keeps executing the statement while the condition is true.

Table 1 Growth of an Investment

Year	Balance
0	$10,000.00
1	$10,500.00
2	$11,025.00
3	$11,576.25
4	$12,155.06
5	$12,762.82

Most commonly, the statement is a block statement, that is, a set of statements delimited by `{ }`.

In our case, we want to know when the bank account has reached a particular balance. While the balance is less, we keep adding interest and incrementing the `year` counter:

```
while (balance < targetBalance)
{
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

Here is the program that solves our investment problem:

ch06/invest1/Investment.java

```
/**
   A class to monitor the growth of an investment that
   accumulates interest at a fixed annual rate.
```

```
*/
public class Investment
{
   /**
      Constructs an Investment object from a starting balance and
      interest rate.
         @param aBalance the starting balance
         @param aRate the interest rate in percent
   */
   public Investment(double aBalance, double aRate)
   {
        balance = aBalance;
        rate = aRate;
        years = 0;
   }

```

```
   /**
      Keeps accumulating interest until a target balance has
      been reached.
         @param targetBalance the desired balance
   */
   public void waitForBalance(double targetBalance)
   {
      while (balance < targetBalance)
      {
            years++;
            double interest = balance * rate / 100;
            balance = balance + interest;
      }
   }

   /**
      Gets the current investment balanace.
         @return the current balance
   */
   public double getBalance()
   {
         return balance;
   }

```

```
    /**
    Gets the number of years this investment has accumulated
    interest.
          @return the number of years since the start of the
investment
    */
    public int getYears()
    {
          return years;
    }

    private double balance;
    private double rate;
    private int years;
  }
```

ch06/invest1/InvestmentRunner.java

```
    /**
    This program computes how long it takes for an investment
    to double.
    */
    public class InvestmentRunner
    {
         public static void main(String[] args)
         {
               final double INITIAL_BALANCE =
10000;
               final double RATE = 5;
               Investment invest = new
Investment(INITIAL_BALANCE, RATE);
               Invest.waitForBalance(2 *
INITIAL_BALANCE);
               int years = invest.getYears();
               System.out.printf("The investment
doubled after"
                     + years + " years"
         }
  }
```

Output

```
The investment doubled after 15 years
```

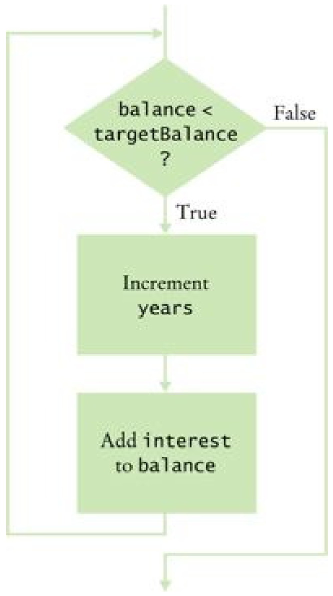
A `while` statement is often called a *loop*. If you draw a flowchart, you will see that the control loops backwards to the test after every iteration (see Figure 1).

The following loop,

```
while (true)
      statement
```

executes the statement over and over, without terminating. Whoa! Why would you want that? The program would never stop. There are two reasons. Some programs indeed never stop; the software controlling an automated teller machine, a telephone switch, or a microwave oven doesn't ever stop (at least not until the device is turned off). Our programs aren't usually of that kind, but even if you can't terminate the loop, you can exit from the method that contains it. This can be helpful when the termination test naturally falls in the middle of the loop (see Advanced Topic 6.3).

Figure 1



Flowchart of a `while` Loop

SYNTAX 6.1 The `while` Statement

`while` (*condition*)
   statement

Example:

```
while (balance < targetBalance)
{
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

Purpose:

To repeatedly execute a statement as long as a condition is true

SELF CHECK

1. How often is the following statement in the loop executed?

   `while (false)` *`statement`*`;`

2. What would happen if `RATE` was set to 0 in the `main` method of the `InvestmentRunner` program?

COMMON ERROR 6.1: Infinite Loops

The most annoying loop error is an infinite loop: a loop that runs forever and can be stopped only by killing the program or restarting the computer. If there are output statements in the loop, then reams and reams of output flash by on the screen. Otherwise, the program just sits there and hangs, seeming to do nothing. On some systems you can kill a hanging program by hitting Ctrl+Break or Ctrl+C. On others, you can close the window in which the program runs.

A common reason for infinite loops is forgetting to advance the variable that controls the loop:

```
int years = 0;
```

```
while (years < 20)
{
      double interest = balance * rate / 100;
      balance = balance + interest;
}
```

Here the programmer forgot to add a statement for incrementing `years` in the loop. As a result, the value of `years` always stays `0`, and the loop never comes to an end.

Another common reason for an infinite loop is accidentally incrementing a counter that should be decremented (or vice versa). Consider this example:

```
int years = 20;
while (years > 0)
{
    years++; // Oops, should have been years--
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

The `years` variable really should have been decremented, not incremented. This is a common error, because incrementing counters is so much more common than decrementing that your fingers may type the `++` on autopilot. As a consequence, `years` is always larger than `0`, and the loop never terminates. (Actually, `years` eventually will exceed the largest representable positive integer and wrap around to a negative number. Then the loop exits—of course, that takes a long time, and the result is completely wrong.)

COMMON ERROR 6.2: Off-by-One Errors

Consider our computation of the number of years that are required to double an investment:

```
int years = 0;
while (balance < 2 * initialBalance)
{
     years++;
     double interest = balance * rate / 100;
     balance = balance + interest;
}
System.out.println(
          "The investment reached the target after
"
          + years + " years.");
```

Should `years` start at 0 or at 1? Should you test for `balance < 2 * initialBalance` or for `balance <= 2 * initialBalance`? It is easy to be *off by one* in these expressions.

Some people try to solve *off-by-one errors* by randomly inserting `+1` or `−1` until the program seems to work. That is, of course, a terrible strategy. It can take a long time to compile and test all the various possibilities. Expending a small amount of mental effort is a real time saver.

Fortunately, off-by-one errors are easy to avoid, simply by thinking through a couple of test cases and using the information from the test cases to come up with a rationale for the correct loop condition.

An off-by-one error is a common error when programming loops. Think through simple test cases to avoid this type of error.

Should `years` start at 0 or at 1? Look at a scenario with simple values: an initial balance of $100 and an interest rate of 50%. After year 1, the balance is $150, and after year 2 it is $225, or over $200. So the investment doubled after 2 years. The loop executed two times, incrementing `years` each time. Hence `years` must start at 0, not at 1.

In other words, the `balance` variable denotes the balance *after* the end of the year. At the outset, the `balance` variable contains the balance after year 0 and not after year 1.

233

234

Next, should you use a < or <= comparison in the test? That is harder to figure out, because it is rare for the balance to be exactly twice the initial balance. Of course, there is one case when this happens, namely when the interest is 100%. The loop executes once. Now `years` is 1, and `balance` is exactly equal to `2 * initialBalance`. Has the investment doubled after one year? It has. Therefore, the loop should *not* execute again. If the test condition is `balance < 2 * initialBalance`, the loop stops, as it should. If the test condition had been `balance <= 2 * initialBalance`, the loop would have executed once more.

In other words, you keep adding interest while the balance *has not yet doubled*.

ADVANCED TOPIC 6.1: `do` Loops

Sometimes you want to execute the body of a loop at least once and perform the loop test after the body was executed. The `do` loop serves that purpose:

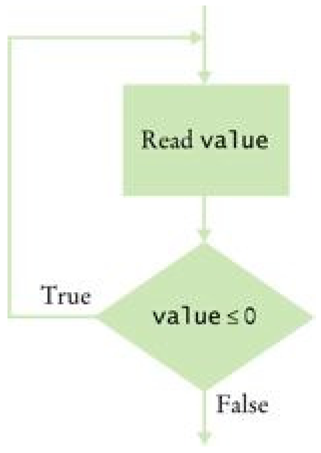
```
do
   statement
while (condition);
```

The *statement* is executed while the *condition* is true. The condition is tested after the statement is executed, so the statement is executed at least once.

For example, suppose you want to make sure that a user enters a positive number. As long as the user enters a negative number or zero, just keep prompting for a correct input. In this situation, a `do` loop makes sense, because you need to get a user input before you can test it.

```
double value;
do
{
    System.out.print("Please enter a positive
number: ");
    value = in.nextDouble();
}
while (value <= 0);
```

The figure shows a flowchart of this loop.



Flowchart of a `do` Loop

In practice, this situation is not very common. You can always replace a `do` loop with a `while` loop, by introducing a `boolean` control variable.

```
    boolean done = false;
    while (!done)
    {
        System.out.print("Please enter a positive
    number: ");
        value = in.nextDouble();
        if (value > 0) done = true;
    }
```

RANDOM FACT 6.1: Spaghetti Code

In this chapter we are using flowcharts to illustrate the behavior of the loop statements. It used to be common to draw flowcharts for every method, on the theory that flowcharts were easier to read and write than the actual code (especially in the days of machine-language and assembler programming). Flowcharts are no longer routinely used for program development and documentation.

Flowcharts have one fatal flaw. Although it is possible to express the `while` and `do` loops with flowcharts, it is also possible to draw flowcharts that cannot be programmed with loops. Consider the chart in the Spaghetti Code figure. The top of the flowchart is simply a statement

```
years = 1;
```

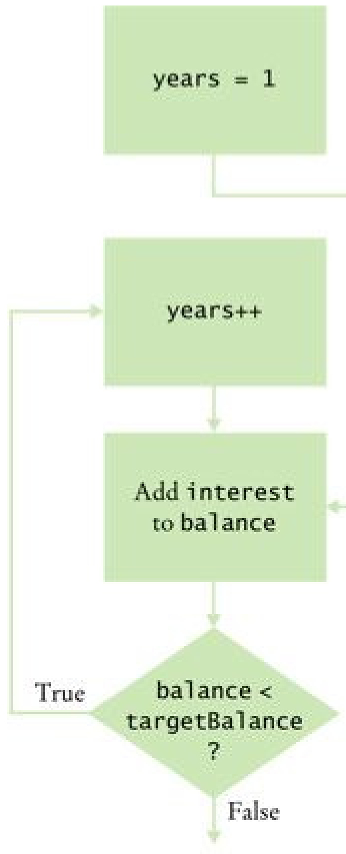
The lower part is a do loop:

```
do
{
   years++;
   double interest = balance * rate / 100;
   balance = balance + interest;
}
while (balance < targetBalance);
```

But how can you join these two parts? According to the flowchart, you are supposed to jump from the first statement into the middle of the loop, skipping the first statement.

```
years = 1;
goto a; // Not an actual Java statement
do
{
     years++;
     a:
     double interest = balance * rate / 100;
     balance = balance + interest;
}
while (balance < targetBalance);
```

235



Spaghetti Code

Spaghetti Code

In fact, why even bother with the `do` loop? Here is a faithful interpretation of the flowchart:

```
years = 1;
goto a; // Not an actual Java statement
b:
years++;
a:
double interest = balance * rate / 100;
balance = balance + interest;
if (balance < targetBalance) goto b;
```

This nonlinear control flow turns out to be extremely hard to read and understand if you have more than one or two `goto` statements. Because the lines denoting the `goto` statements weave back and forth in complex flowcharts, the resulting code is named *spaghetti code*.

In 1968 the influential computer scientist Edsger Dijkstra wrote a famous note, entitled "Goto Statements Considered Harmful" [1], in which he argued for the use of loops instead of unstructured jumps. Initially, many programmers who had been using `goto` for years were mortally insulted and promptly dug out examples in which the use of `goto` led to clearer or faster code. Some languages offer weaker forms of `goto` that are less harmful, such as the `break` statement in Java, discussed in Advanced Topic 6.4. Nowadays, most computer scientists accept Dijkstra's argument and fight bigger battles than optimal loop design.

6.2 for Loops

One of the most common loop types has the form

```
i = start;
while (i <= end)
{
      . . .
      i++;
}
```

Because this loop is so common, there is a special form for it that emphasizes the pattern:

```
for (i = start; i <= end; i++)
{
      . . .
}
```

You can also declare the loop counter variable inside the `for` loop header. That convenient shorthand restricts the use of the variable to the body of the loop (as will be discussed further in Advanced Topic 6.2).

```
for (int i = start; i <= end; i++)
{
       . . .
```

```
    }
```

Let us use this loop to find out the size of our $10,000 investment if 5% interest is compounded for 20 years. Of course, the balance will be larger than $20,000, because at least $500 is added every year. You may be surprised to find out just how much larger the balance is.

SYNTAX 6.2 The `for` Statement

`for` (*initialization*; *condition*; *update*)
  statement

Example:

```
for (i = 1; i <= n; i++)
{
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

Purpose:

To execute an initialization, then keep executing a statement and updating an expression while a condition is true

In our loop, we let `i` go from `1` to `n`, the number of years for which we want to compound interest.

You use a `for` loop when a variable runs from a starting to an ending value with a constant increment or decrement.

```
for (int i = 1; i <= n; i++)
{
     double interest = balance * rate / 100;
     balance = balance + interest;
}
```

Figure 2 shows the corresponding flowchart.

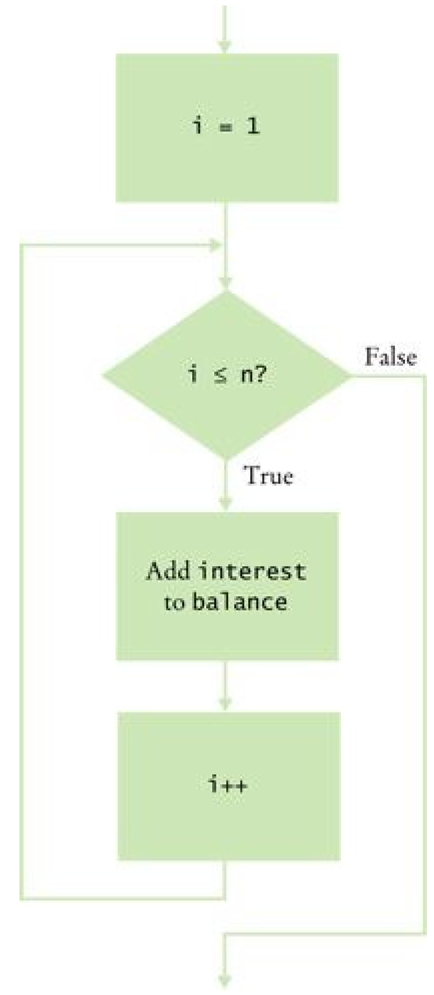
The three slots in the `for` header can contain any three expressions. You can count down instead of up:

```
for (years = n; years > 0; years--)
```

The increment or decrement need not be in steps of 1:

```
for (x = -10; x <= 10; x = x + 0.5) . . .
```

Figure 2



Flowchart of a `for` Loop

238

239

It is possible—but a sign of unbelievably bad taste—to put unrelated conditions into the loop header:

```
for (rate = 5; years-- > 0;
System.out.println(balance))
      . . . // Bad taste
```

We won't even begin to decipher what that might mean. You should stick with `for` loops that initialize, test, and update a single variable.

ch06/invest2/Investment.java

```
/**
A class to monitor the growth of an investment that
accumulates interest at a fixed annual rate.
*/
public class Investment
{
     /**
       Constructs an Investment object from a starting balance and
       interest rate.
             @param aBalance the starting balance
             @param aRate the interest rate in percent
     */
     public Investment(double aBalance, double
aRate)
     {
             balance = aBalance;
             rate = aRate;
             years = 0;
     }

    /**
       Keeps accumulating interest until a target balance has
       been reached.
             @param targetBalance the desired balance
    */
    public void waitForBalance(double
targetBalance)
    {
            while (balance < targetBalance)
            {
                  years++;
```

```
                double interest = balance * rate
/ 100;
                balance = balance + interest;
            }
    }

    /**
    Keeps accumulating interest for a given number of years.
            @param n the number of years
    */
    public void waitYears(int n)
    {
            for (int i = 1; i <= n; i++)
            {
                double interest = balance * rate
/ 100;
                balance = balance + interest;
```

```
            }
            years = years + n;
    }

    /**
    Gets the current investment balance.
            @return the current balance
    */
    public double getBalance()
      {
            return balance;
    }

    /**
    Gets the number of years this investment has accumulated
    interest.
            @return the number of years since the start of the
investment
    */
    public int getYears()
    {
            return years;
    }

    private double balance;
    private double rate;
```

```
   private int years;
}
```

ch06/invest2/InvestmentRunner.java

```
/**
   This program computes how much an investment grows in
   a given number of years.
*/
public class InvestmentRunner
{
   public static void main(String[] args)
   {
      final double INITIAL_BALANCE = 10000;
      final double RATE = 5;
      final int YEARS = 20;
      Investment invest = new Investment(INITIAL_BALANCE, RATE);
      invest.waitYears(YEARS);
      double balance = invest.getBalance();
      System.out.printf("The balance after %d years is %.2f\n",
            YEARS, balance);
   }
}
```

Output

```
The balance after 20 years is 26532.98
```

240

241

SELF CHECK

3. Rewrite the `for` loop in the `waitYears` method as a `while` loop.

4. How many times does the following `for` loop execute?

```
for (i = 0; i <= 10; i++)
    System.out.println(i * i);
```

QUALITY TIP 6.1: Use `for` Loops for Their Intended Purpose

A `for` loop is an *idiom* for a `while` loop of a particular form. A counter runs from the start to the end, with a constant increment:

```
for (set counter to start; test whether counter at
end;
        update counter by increment)
{ . . .
   // counter, start, end, increment not changed here
}
```

If your loop doesn't match this pattern, don't use the `for` construction. The compiler won't prevent you from writing idiotic `for` loops:

```
// Bad style-unrelated header expressions
for (System.out.println("Inputs:");
         (x = in.nextDouble()) > 0;
         sum = sum + x)
   count++;
for (int i = 1; i <= years; i++)
{
  // Bad style-modifies counter
   if (balance >= targetBalance)
      i = years + 1;
   else
   {
     double interest = balance * rate / 100;
     balance = balance + interest;
   }
}
```

These loops will work, but they are plainly bad style. Use a `while` loop for iterations that do not fit the `for` pattern.

241

Common Error 6.3: Forgetting a Semicolon

Occasionally all the work of a loop is already done in the loop header. Suppose you ignored Quality Tip 6.1; then you could write an investment doubling loop as follows:

```
for (years = 1;
       (balance = balance + balance * rate / 100)
< targetBalance;
       years++)
    ;
System.out.println(years);
```

The body of the for loop is completely empty, containing just one empty statement terminated by a semicolon.

If you do run into a loop without a body, it is important that you make sure the semicolon is not forgotten. If the semicolon is accidentally omitted, then the next line becomes part of the loop statement!

```
for (years = 1;
        (balance = balance + balance * rate / 100)
< targetBalance;
        years++)
System.out.println(years);
```

You can avoid this error by using an empty block `{ }` instead of an empty statement.

Common Error 6.4: A Semicolon Too Many

What does the following loop print?

```
sum = 0;
for (i = 1; i <= 10; i++);
   sum = sum + i;
System.out.println(sum);
```

Of course, this loop is supposed to compute 1 + 2 + … + 10 = 55. But actually, the print statement prints 11!

Why 11? Have another look. Did you spot the semicolon at the end of the `for` loop header? This loop is actually a loop with an empty body.

```
for (i = 1; i <= 10; i++)
   ;
```

The loop does nothing 10 times, and when it is finished, `sum` is still 0 and `i` is 11. Then the statement

```
sum = sum + i;
```

is executed, and `sum` is 11. The statement was indented, which fools the human reader. But the compiler pays no attention to indentation.

Of course, the semicolon at the end of the statement was a typing error. Someone's fingers were so used to typing a semicolon at the end of every line that a semicolon was added to the `for` loop by accident. The result was a loop with an empty body. 242

243

QUALITY TIP 6.2: Don't Use != to Test the End of a Range

Here is a loop with a hidden danger:

```
for (i = 1; i != n; i++)
```

The test `i != n` is a poor idea. How does the loop behave if `n` happens to be zero or negative?

The test `i != n` is never false, because `i` starts at 1 and increases with every step.

The remedy is simple. Use <= rather than `!=` in the condition:

```
for (i = 1; i <= n; i++)
```

ADVANCED TOPIC 6.2: Variables Defined in a `for` Loop Header

As mentioned, it is legal in Java to declare a variable in the header of a `for` loop. Here is the most common form of this syntax:

```
for (int i = 1; i <= n; i++)
{
   . . .
}
// i no longer defined here
```

The scope of the variable extends to the end of the `for` loop. Therefore, `i` is no longer defined after the loop ends. If you need to use the value of the variable beyond the end of the loop, then you need to define it outside the loop. In this loop, you don't need the value of `i`—you know it is `n + 1` when the loop is finished. (Actually, that is not quite true—it is possible to break out of a loop before its end; see Advanced Topic 6.4). When you have two or more exit conditions, though, you may still need the variable. For example, consider the loop

```
for (i = 1; balance < targetBalance && i <= n; i++)
{
   . . .
}
```

You want the balance to reach the target, but you are willing to wait only a certain number of years. If the balance doubles sooner, you may want to know the value of `i`. Therefore, in this case, it is not appropriate to define the variable in the loop header.

Note that the variables named `i` in the following pair of `for` loops are independent:

```
for (int i = 1; i <= 10; i++)
   System.out.println(i * i);
for (int i = 1; i <= 10; i++) // Declares a new variable i
   System.out.println(i * i * i);
```

In the loop header, you can declare multiple variables, as long as they are of the same type, and you can include multiple update expressions, separated by commas:

```
for (int i = 0, j = 10; i <= 10; i++, j--)
{
   . . .
}
```

However, many people find it confusing if a `for` loop controls more than one variable. I recommend that you not use this form of the `for` statement (see Quality

Tip 6.1). Instead, make the `for` loop control a single counter, and update the other variable explicitly:

```
int j = 10;
for (int i = 0; i <= 10; i++)
{
   . . .
   j--;
}
```

6.3 Nested Loops

Sometimes, the body of a loop is again a loop. We say that the inner loop is *nested* inside an outer loop. This happens often when you process two-dimensional structures, such as tables.

Loops can be nested. A typical example of nested loops is printing a table with rows and columns.

Let's look at an example that looks a bit more interesting than a table of numbers. We want to generate the following triangular shape:

```
[]
[][]
[][][]
[][][][]
[][][][][]
[][][][][][]
[][][][][][][]
```

The basic idea is simple. We generate a sequence of rows:

```
for (int i = 1; i <= width; i++)
{
   // Make triangle row
 ...
}
```

How do you make a triangle row? Use another loop to concatenate the squares `[]` for that row. Then add a newline character at the end of the row. The `i`th row has `i` symbols, so the loop counter goes from 1 to `i`.

```
    for (int j = 1; j <= i; j++)
       r = r + "[]";
    r = r + "\n";
```

Putting both loops together yields two *nested loops*:

```
    String r = "";
    for (int i = 1; i <= width; i++)
    {
       // Make triangle row
       for (int j = 1; j <= i; j++)
          r = r + "[]";
       r = r + "\n";
    }
    return r;
```

Here is the complete program:

ch06/triangle1/Triangle.java

```
/**
   This class describes triangle objects that can be displayed
   as shapes like this:
      []
      [][]
      [][][].
*/
public class Triangle
{
   /**
      Constructs a triangle.
      @param aWidth the number of [] in the last row of the triangle
   */
   public Triangle(int aWidth)
   {
      width = aWidth;
   }

   /**
      Computes a string representing the triangle.
      @return a string consisting of [] and newline characters
```

```
     */
    public String toString()
    {
          String r = “”;
          for (int i = 1; i <= width; i++)
          {
                // Make triangle row
                for (int j = 1; j <= i; j++)
                      r = r + “[]”;
                r = r + “\n”;
          }
          return r;
    }

     private int width;
  }
```

ch06/triangle1/TriangleRunner.java

```
/**
 This program prints two triangles.
*/
public class TriangleRunner
{
    public static void main(String[] args)
    {
          Triangle small = new Triangle(3);
          System.out.println(small.toString());

          Triangle large = new Triangle(15);
          System.out.println(large.toString());
    }
}
```

Output

```
[]
[][]
[][][]

[]
[][]
[][][]
[][][][]
[][][][][]
[][][][][][]
[][][][][][][]
[][][][][][][][]
[][][][][][][][][]
[][][][][][][][][][]
[][][][][][][][][][][]
[][][][][][][][][][][][]
[][][][][][][][][][][][][]
[][][][][][][][][][][][][][]
[][][][][][][][][][][][][][][]
```

SELF CHECK

5. How would you modify the nested loops so that you print a square instead of a triangle?

6. What is the value of `n` after the following nested loops?

```
int n = 0;
for (int i = 1; i <= 5; i++)
    for (int j = 0; j < i; j++)
        n = n + j;
```

246

247

6.4 Processing Sentinel Values

Suppose you want to process a set of values, for example a set of measurements. Your goal is to analyze the data and display properties of the data set, such as the average or the maximum value. You prompt the user for the first value, then the second value, then the third, and so on. When does the input end?

One common method for indicating the end of a data set is a *sentinel value*, a value that is not part of the data. Instead, the sentinel value indicates that the data has come to an end.

Some programmers choose numbers such as 0 or −1 as sentinel values. But that is not a good idea. These values may well be valid inputs. A better idea is to use an input that is not a number, such as the letter `Q`. Here is a typical program run:

```
Enter value, Q to quit: 1
Enter value, Q to quit: 2
Enter value, Q to quit: 3
Enter value, Q to quit: 4
Enter value, Q to quit: Q
Average = 2.5
Maximum = 4.0
```

Of course, we need to read each input as a string, not a number. Once we have tested that the input is not the letter `Q`, we convert the string into a number.

```
System.out.print("Enter value, Q to quit: ");
String input = in.next();
if (input.equalsIgnoreCase("Q"))
  We are done
else
{
   double x = Double.parseDouble(input);
   . . .
}
```

Now we have another problem. The test for loop termination occurs in the *middle* of the loop, not at the top or the bottom. You must first try to read input before you can test whether you have reached the end of input. In Java, there isn't a ready–made control structure for the pattern “do work, then test, then do more work”. Therefore, we use a combination of a `while` loop and a `boolean` variable.

Sometimes, the termination condition of a loop can only be evaluated in the middle of a loop. You can introduce a Boolean variable to control such a loop.

```
boolean done = false;
while (!done)
{
  Print prompt
      String input = read input;
      if (end of input indicated)
         done = true;
```

```
            else
            {
         Process input
            }
    }
```

This pattern is sometimes called “loop and a half”. Some programmers find it clumsy to introduce a control variable for such a loop. Advanced Topic 6.3 shows several alternatives.

Let's put together the data analysis program. To decouple the input handling from the computation of the average and the maximum, we'll introduce a class `DataSet`. You add values to a `DataSet` object with the `add` method. The `getAverage` method returns the average of all added data and the `getMaximum` method returns the largest.

ch06/dataset/DataAnalyzer.java

```
import java.util.Scanner;

/**
   This program computes the average and maximum of a set
   of input values.
*/
public class DataAnalyzer
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      DataSet data = new DataSet();

      boolean done = false;
      while (!done)
      {
         System.out.print(“Enter value, Q to quit: ”);
         String input = in.next();
         if (input.equalsIgnoreCase(“Q”))
            done = true;
         else
         {
```

```
                        double x = Double.parseDouble(input);
                        data.add(x);
                    }
              }

              System.out.println("Average = " + data.getAverage());
              System.out.println("Maximum = " + data.getMaximum());
        }
  }
```

ch06/dataset/DataSet.java

```
 /**
    Computes information about a set of data values.
 */
 public class DataSet
 {
```

248

249

```
      /**
         Constructs an empty data set.
      */
      public DataSet()
    {
          sum = 0;
          count = 0;
          maximum = 0;
      }

      /**
         Adds a data value to the data set.
            @param x a data value
      */
      public void add(double x)
      {
           sum = sum + x;
           if (count == 0 || maximum < x) maximum = x;
           count++;
      }

      /**
```

```
        Gets the average of the added data.
                @return the average or 0 if no data has been added
        */
        public double getAverage()
        {
                if (count == 0) return 0;
                else return sum / count;
        }

        /**
        Gets the largest of the added data.
                @return the maximum or 0 if no data has been
added
        */
        public double getMaximum()
        {
                return maximum;
         }

         private double sum;
         private double maximum;
         private int count;
    }
```

Output

```
Enter value, Q to quit: 10
Enter value, Q to quit: 0
Enter value, Q to quit: -1
Enter value, Q to quit: Q
Average = 3.0
Maximum = 10.0
```

Self Check

7. Why does the `DataAnalyzer` class call `in.next` and not `in.nextDouble`?

8. Would the `DataSet` class still compute the correct maximum if you simplified the update of the `maximum` field in the `add` method to the following statement?

```
            if (maximum < x) maximum = x;
```

How To 6.1: Implementing Loops

You write a loop because your program needs to repeat an action multiple times. As you have seen in this chapter, there are several loop types, and it isn't always obvious how to structure loop statements. This How To walks you through the thought process that is involved when programming a loop.

Step 1 List the work that needs to be done in every step of the loop body.

For example, suppose you need to read in input values in gallons and convert them to liters until the end of input is reached. Then the operations are:

- Read input.
- Convert the input to liters.
- Print out the response.

Suppose you need to scan through the characters of a string and count the vowels. Then the operations are:

- Get the next character.
- If it's a vowel, increase a counter.

Step 2 Find out how often the loop is repeated.

Typical answers might be:

- Ten times
- Once for each character in the string
- Until the end of input is reached
- While the balance is less than the target balance

If a loop is executed for a definite number of times, a `for` loop is usually appropriate. The first two answers above lead to `for` loops, such as

```
for (int i = 1; i <= 10; i++) . . .
for (int i = 0; i < str.length(); i++) . . .
```

The next two need to be implemented as `while` loops—you don't know how many times the loop body is going to be repeated.

Step 3 With a `while` loop, find out where you can determine that the loop is finished.

There are three possibilities:

- Before entering the loop
- In the middle of the loop
- At the end of the loop

For example, if you execute a loop while the balance is less than the target balance, you can check for that condition at the beginning of the loop. If the balance is less than the target balance, you enter the loop. If not, you are done. In such a case, your loop has the form

```
while (condition)
{
   Do work
}
```

However, checking for input requires that you first *read* the input. That means, you'll need to enter the loop, read the input, and then decide whether you want to go any further. Then your loop has the form

```
boolean done = false;
while (!done)
{
   Do the work needed to check the condition
      if (condition)
         done = true;
      else
      {
     Do more work
      }
}
```

This loop structure is sometimes called a *"loop and a half"*.

Finally, if you know whether you need to go on after you have gone through the loop once, then you use a `do/while` loop:

```
do
{
   Do work
}
while (condition)
```

However, these loops are very rare in practice.

Step 4 Implement the loop by putting the operations from Step 1 into the loop body.

When you write a `for` loop, you usually use the loop index inside the loop body. For example, "get the next character" is implemented as the statement

```
char ch = str.charAt(i);
```

Step 5 Double-check your variable initializations.

If you use a Boolean variable `done`, make sure it is initialized to `false`. If you accumulate a result in a `sum` or `count` variable, make sure that you set it to 0 before entering the loop for the first time.

Step 6 Check for off-by-one errors.

Consider the simplest possible scenarios:

- If you read input, what happens if there is no input at all? Exactly one input?
- If you look through the characters of a string, what happens if the string is empty? If it has one character in it?
- If you accumulate values until some target has been reached, what happens if the target is 0? A negative value?

Manually walk through every instruction in the loop, including all initializations. Carefully check all conditions, paying attention to the difference between comparisons such as < and <=. Check that the loop is not traversed at all, or only once, and that the final result is what you expect.

If you write a `for` loop, check to see whether your bounds should be symmetric or asymmetric (see Quality Tip 6.3), and count the number of iterations (see Quality Tip 6.4).

QUALITY TIP 6.3: Symmetric and Asymmetric Bounds

It is easy to write a loop with `i` going from 1 to `n`:

```
for (i = 1; i <= n; i++) . . .
```

The values for `i` are bounded by the relation 1 ≤ `i` ≤ `n`. Because there are ≤ comparisons on both bounds, the bounds are called *symmetric*.

When traversing the characters in a string, the bounds are *asymmetric*.

```
for (i = 0; i < str.length(); i++) . . .
```

The values for `i` are bounded by 0 ≤ `i` < `str.length()`, with a ≤ comparison to the left and a < comparison to the right. That is appropriate, because `str.length()` is not a valid position.

Make a choice between symmetric and asymmetric loop bounds.

It is not a good idea to force symmetry artificially:

```
for (i = 0; i <= str.length() - 1; i++) . . .
```

That is more difficult to read and understand.

For every loop, consider which form is most natural for the problem, and use that.

QUALITY TIP 6.4: Count Iterations

Finding the correct lower and upper bounds for an iteration can be confusing. Should I start at 0? Should I use `<= b` or `< b` as a termination condition?

Count the number of iterations to check that your `for` loop is correct.

Counting the number of iterations is a very useful device for better understanding a loop. Counting is easier for loops with asymmetric bounds. The loop

```
for (i = a; i < b; i++) . . .
```

is executed `b - a` times. For example, the loop traversing the characters in a string,

```
for (i = 0; i < str.length(); i++) . . .
```

runs `str.length()` times. That makes perfect sense, because there are `str.length()` characters in a string.

The loop with symmetric bounds,

```
for (i = a; i <= b; i++)
```

is executed `b - a + 1` times. That "`+ 1`" is the source of many programming errors. For example,

```
for (n = 0; n <= 10; n++)
```

runs 11 times. Maybe that is what you want; if not, start at `1` or use `< 10`.

One way to visualize this "`+ 1`" error is to think of the posts and sections of a fence. Suppose the fence has ten sections (=). How many posts (|) does it have?

```
|=|=|=|=|=|=|=|=|=|=|
```

A fence with ten sections has *eleven* posts. Each section has one post to the left, *and* there is one more post after the last section. Forgetting to count the last iteration of a "<=" loop is often called a "fence post error".

If the increment is a value `c` other than 1, and `c` divides `b - a`, then the counts are

`(b - a) / c`	for the asymmetric loop
`(b - a) / c + 1`	for the symmetric loop

For example, the loop `for (i = 10; i <= 40; i += 5)` executes (40 – 10)/5 + 1 = 7 times.

ADVANCED TOPIC 6.3: The “Loop and a Half” Problem

Reading input data sometimes requires a loop such as the following, which is somewhat unsightly:

```
boolean done = false;
while (!done)
{
      String input = in.next();
      if (input.equalsIgnoreCase(“Q”))
         done = true;
      else
      {
    Process data
      }
}
```

The true test for loop termination is in the middle of the loop, not at the top. This is called a “loop and a half”, because one must go halfway into the loop before knowing whether one needs to terminate.

Some programmers dislike the introduction of an additional Boolean variable for loop control. Two Java language features can be used to alleviate the “loop and a half” problem. I don't think either is a superior solution, but both approaches are fairly common, so it is worth knowing about them when reading other people's code.

You can combine an assignment and a test in the loop condition:

```
while (!(input = in.next()).equalsIgnoreCase(“Q”))
{
  Process data
}
```

The expression

```
(input = in.next()).equalsIgnoreCase("Q")
```

means, “First call in.next(), then assign the result to input, then test whether it equals “Q””. This is an expression with a side effect. The primary purpose of the expression is to serve as a test for the while loop, but it also does some work—namely, reading the input and storing it in the variable input. In general, it is a bad idea to use side effects, because they make a program hard to read and maintain. In this case, however, that practice is somewhat seductive, because it eliminates the control variable done, which also makes the code hard to read and maintain.

The other solution is to exit the loop from the middle, either by a return statement or by a break statement (see Advanced Topic 6.4).

```
public void processInput(Scanner in)
{
   while (true)
   {
        String input = in.next();
        if (input.equalsIgnoreCase(“Q”))
            return;
   Process data
   }
}
```

Advanced Topic 6.4: The break and continue Statements

You already encountered the break statement in Advanced Topic 5.2, where it was used to exit a switch statement. In addition to breaking out of a switch statement, a break statement can also be used to exit a while, for, or do loop. For example, the break statement in the following loop terminates the loop when the end of input is reached.

```
while (true)
{
   String input = in.next();
   if (input.equalsIgnoreCase("Q"))
      break;
   double x = Double.parseDouble(input);
   data.add(x);
}
```

In general, a `break` is a very poor way of exiting a loop. In 1990, a misused `break` caused an AT&T 4ESS telephone switch to fail, and the failure propagated through the entire U.S. network, rendering it nearly unusable for about nine hours. A programmer had used a `break` to terminate an `if` statement. Unfortunately, `break` cannot be used with `if`, so the program execution broke out of the enclosing `switch` statement, skipping some variable initializations and running into chaos [2, p. 38]. Using `break` statements also makes it difficult to use *correctness proof* techniques (see Advanced Topic 6.5).

However, when faced with the bother of introducing a separate loop control variable, some programmers find that `break` statements are beneficial in the "loop and a half" case. This issue is often the topic of heated (and quite unproductive) debate. In this book, we won't use the `break` statement, and we leave it to you to decide whether you like to use it in your own programs.

In Java, there is a second form of the `break` statement that is used to break out of a nested statement. The statement `break` *label*; immediately jumps to the *end* of the statement that is tagged with a label. Any statement (including `if` and block statements) can be tagged with a label—the syntax is

```
label: statement
```

The labeled `break` statement was invented to break out of a set of nested loops.

```
outerloop:
while (outer loop condition)
{     . . .
      while (inner loop condition)
      {     . . .
            if (something really bad happened)
               break outerloop;
      }
}
Jumps here if something really bad happened
```

Naturally, this situation is quite rare. We recommend that you try to introduce additional methods instead of using complicated nested loops.

Finally, there is another `goto`– like statement, the `continue` statement, which jumps to the end of the *current iteration* of the loop. Here is a possible use for this statement:

```
while (!done)
{
      String input = in.next();
      if (input.equalsIgnoreCase("Q"))
      {
         done = true;
         continue; // Jump to the end of the loop body
      }
      double x = Double.parseDouble(input);
      data.add(x);
      // continue statement jumps here
}
```

By using the `continue` statement, you don't need to place the remainder of the loop code inside an `else` clause. This is a minor benefit. Few programmers use this statement.

6.5 Random Numbers and Simulations

In a simulation you generate random events and evaluate their outcomes. Here is a typical problem that can be decided by running a simulation: the *Buffon needle experiment*, devised by Comte Georges– Louis Leclerc de Buffon (1707–1788), a French naturalist. On each *try*, a one–inch long needle is dropped onto paper that is ruled with lines 2 inches apart. If the needle drops onto a line, count it as a hit. (See Figure 3.) Buffon conjectured that the quotient *tries/hits* approximates π.

In a simulation, you repeatedly generate random numbers and use them to simulate an activity.

Now, how can you run this experiment in the computer? You don't actually want to build a robot that drops needles on paper. The `Random` class of the Java library implements a *random number generator*, which produces numbers that appear to be completely random. To generate random numbers, you construct an object of the `Random` class, and then apply one of the following methods:

Method	Returns
nextInt(n)	A random integer between the integers 0 (inclusive) and n (exclusive)
nextDouble()	A random floating–point number between 0) (inclusive) and 1 (exclusive)

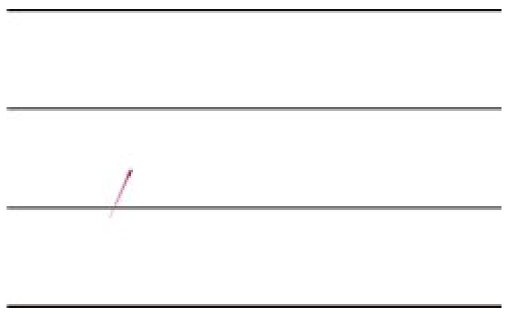
For example, you can simulate the cast of a die as follows:

```
Random generator = new Random();
int d = 1 + generator.nextInt(6);
```

The call `generator.nextInt(6)` gives you a random number between 0 and 5 (inclusive). Add 1 to obtain a number between 1 and 6.

To give you a feeling for the random numbers, run the following program a few times.

Figure 3



The Buffon Needle Experiment

ch06/random1/Die.java

```
import java.util.Random;

/**
   This class models a die that, when cast, lands on a random
   face.
*/
public class Die
{
   /**
      Constructs a die with a given number of sides.
```

```
            @param s the number of sides, e.g., 6 for a normal die
      */
      public Die(int s)
      {
            sides = s;
            generator = new Random();
      }

      /**
      Simulates a throw of the die.
            @return the face of the die
      */
      public int cast()
      {
          return 1 + generator.nextInt(sides);
    }

      private Random generator;
      private int sides;
}
```

ch06/random1/DieSimulator.java

```
/**
This program simulates casting a die ten times.
*/
public class DieSimulator
{
     public static void main(String[] args)
     {
           Die d = new Die(6);
           final int TRIES = 10;
          for (int i = 1; i <= TRIES; i++)
          {
               int n = d.cast();
               System.out.print(n + " ");
          }
        System.out.println();
     }
}
```

Typical Output

```
6 5 6 3 2 6 3 4 4 1
```

Typical Output (Second Run)

```
3 2 2 1 6 5 3 4 1 2
```

As you can see, this program produces a different stream of simulated die casts every time it is run.

Actually, the numbers are not completely random. They are drawn from very long sequences of numbers that don't repeat for a long time. These sequences are computed from fairly simple formulas; they just behave like random numbers. For that reason, they are often called *pseudorandom* numbers. Generating good sequences of numbers that behave like truly random sequences is an important and well–studied problem in computer science. We won't investigate this issue further, though; we'll just use the random numbers produced by the `Random` class.

To run the Buffon needle experiment, we have to work a little harder. When you throw a die, it has to come up with one of six faces. When throwing a needle, however, there are many possible outcomes. You must generate *two* random numbers: one to describe the starting position and one to describe the angle of the needle with the *x*–axis. Then you need to test whether the needle touches a grid line. Stop after 10,000 tries.

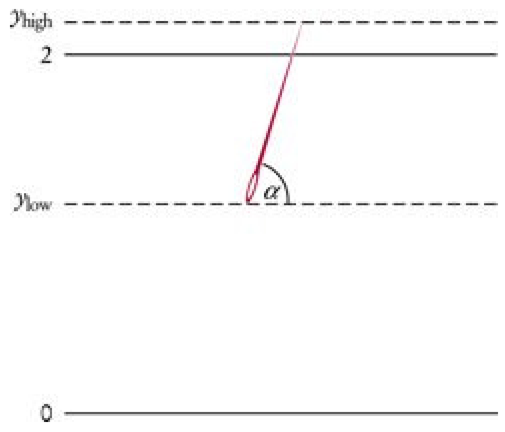
Let us agree to generate the *lower* point of the needle. Its *x*–coordinate is irrelevant, and you may assume its *y* – coordinate y_{low} to be any random number between 0 and 2. However, because it can be a random *floating−point* number, we use the `nextDouble` method of the `Random` class. It returns a random floating – point number between 0 and 1. Multiply by 2 to get a random number between 0 and 2.

The angle α between the needle and the *x*–axis can be any value between 0 degrees and 180 degrees. The upper end of the needle has *y*–coordinate

$$y_{\text{high}} = y_{\text{low}} + \sin(\alpha)$$

The needle is a hit if y_{high} is at least 2. See Figure 4.

Figure 4



When Does the Needle Fall on a Line?

258

259

Here is the program to carry out the simulation of the needle experiment.

ch06/random2/Needle.java

```
import java.util.Random;

/**
   This class simulates a needle in the Buffon needle experiment.
*/
public class Needle
{
   /**
      Constructs a needle.
   */
   public Needle()
   {
      hits = 0;
      tries = 0;
      generator = new Random();
   }

   /**
      Drops the needle on the grid of lines and
      remembers whether the needle hit a line.
   */
   public void drop()
   {
         double ylow = 2 * generator.nextDouble();
         double angle = 180 * generator.nextDouble();

  // Computes high point of needle

         double yhigh = ylow + Math.sin(Math.toRadians(angle));
         if (yhigh >= 2) hits++;
         tries++;
    }

    /**
   Gets the number of times the needle hit a line.
          @return the hit count
    */
    public int getHits()
    {
          return hits;
    }

    /**
          Gets the total number of times the needle was dropped.
          @return the try count
   */
   public int getTries()
   {
         return tries;
   }

   private Random generator;
   private int hits;
   private int tries;
}
```

ch06/random2/NeedleSimulator.java

```
/**
   This program simulates the Buffon needle experiment
   and prints the resulting approximations of pi.
*/
public class NeedleSimulator
{
      public static void main(String[] args)
      {
           Needle n = new Needle();
          final int TRIES1 = 10000;
          final int TRIES2 = 1000000;

          for (int i = 1; i <= TRIES1; i++)
              n.drop();
          System.out.printf("Tries = %d, Tries / Hits = %8.5f\n",
                     TRIES1, (double) n.getTries() / n.getHits());

          for (int i = TRIES1 + 1; i <= TRIES2; i++)
              n.drop();
          System.out.printf("Tries = %d, Tries / Hits = %8.5f\n",
                    TRIES2, (double) n.getTries() / n.getHits());
      }
}
```

Output

```
Tries = 10000, Tries / Hits = 3.08928
Tries = 1000000, Tries / Hits = 3.14204
```

The point of this program is not to compute π—there are far more efficient ways to do that. Rather, the point is to show how a physical experiment can be simulated on the computer. Buffon had to physically drop the needle thousands of times and record the results, which must have been a rather dull activity. The computer can execute the experiment quickly and accurately.

Simulations are very common computer applications. Many simulations use essentially the same pattern as the code of this example: In a loop, a large number of sample values are generated, and the values of certain observations are recorded for each sample. When the simulation is completed, the averages, or other statistics of interest from the observed values are printed out.

A typical example of a simulation is the modeling of customer queues at a bank or a supermarket. Rather than observing real customers, one simulates their arrival and their transactions at the teller window or checkout stand in the computer. One can try different staffing or building layout patterns in the computer simply by making changes in the program. In the real world, making many such changes and measuring their effects would be impossible, or at least, very expensive.

Self Check

9. How do you use a random number generator to simulate the toss of a coin?

10. Why is the `NeedleSimulator` program not an efficient method for computing π?

Advanced Topic 6.5: Loop Invariants

Consider the task of computing a^n, where `a` is a floating-point number and `n` is a positive integer. Of course, you can multiply `a . a . … . a`, `n` times, but if `n` is large, you'll end up doing a lot of multiplication. The following loop computes a^n in far fewer steps:

```
double a = . . .;
int n = . . .;
double r = 1;
double b = a;
int i = n;
while (i > 0)
{
      if (i % 2 == 0)  // n is even
      {
          b = b * b;
          i = i / 2;
      }
      else
      {
            r = r * b;
```

```
                i--;
            }
    }
    // Now r equals a to the nth power
```

Consider the case n = 100. The method performs the steps shown in the table below.

Amazingly enough, the algorithm yields exactly a^{100}. Do you understand why? Are you convinced it will work for all values of n? Here is a clever argument to show that the method always computes the correct result. It demonstrates that whenever the program reaches the top of the `while` loop, it is true that

$$r \cdot b^{i} = a^{n}$$

Certainly, it is true the first time around, because `b = a` and `i = n`. Suppose that (I) holds at the beginning of the loop. Label the values of `r`, `b`, and `i` as “old” when entering the loop, and as “new” when exiting the loop. Assume that upon entry

$$r_{\text{old}} \cdot b_{\text{old}}^{i_{\text{old}}} = a^{n}$$

Computing a^{100}

b	i	r
a	100	1
a^2	50	
a^4	25	
	24	a^4
a^8	12	
a^{16}	6	
a^{32}	3	
	2	a^{36}
a^{64}	1	
	0	a^{100}

In the loop you must distinguish two cases: i_{old} even and i_{old} odd. If i_{old} is even, the loop performs the following transformations:

$$r_{\text{new}} = r_{\text{old}}$$

$$b_{\text{new}} = b_{\text{old}}^2$$

$$i_{\text{new}} = i_{\text{old}} / 2$$

Therefore,

$$r_{\text{new}} \cdot b_{\text{new}}^{i_{\text{new}}} = r_{\text{old}} \cdot \left(b_{\text{old}}\right)^{2 \cdot i_{\text{old}} / 2}$$

$$= r_{\text{old}} \cdot b_{\text{old}}^{i_{\text{old}}}$$

$$= a^n$$

On the other hand, if $\mathtt{i}_{\text{old}}$ is odd, then

$$r_{\text{new}} = r_{\text{old}} \cdot b_{\text{old}}$$

$$b_{\text{new}} = b_{\text{old}}$$

$$i_{\text{new}} = i_{\text{old}} - 1$$

Therefore,

$$r_{\text{new}} \cdot b_{\text{new}}^{i_{\text{new}}} = r_{\text{old}} \cdot b_{\text{old}} \cdot b_{\text{old}}^{i_{\text{old}} - 1}$$

$$= r_{\text{old}} \cdot b_{\text{old}}^{i_{\text{old}}}$$

$$= a^n$$

In either case, the new values for `r`, `b`, and `i` fulfill the *loop invariant* (I). So what? When the loop finally exits, (I) holds again:

$$r \cdot b^i = a^n$$

Furthermore, we know that `i = 0`, because the loop is terminating. But because `i = 0`, $r \cdot b^i = r \cdot b^0 = r$. Hence $r = a^n$, and the method really does compute the `n`th power of `a`.

This technique is quite useful, because it can explain an algorithm that is not at all obvious. The condition (I) is called a loop invariant because it is true when the loop is entered, at the top of each pass, and when the loop is exited. If a loop invariant is chosen skillfully, you may be able to deduce correctness of a computation. See [3] for another nice example.

Random Fact 6.2: Correctness Proofs

In Advanced Topic 6.5 we introduced the technique of loop invariants. If you skipped that topic, have a glance at it now. That technique can be used to rigorously prove that a loop computes exactly the value that it is supposed to compute. Such a proof is far more valuable than any testing. No matter how many test cases you try, you always worry whether another case that you haven't tried yet might show a bug. A proof settles the correctness for *all possible inputs*.

For some time, programmers were very hopeful that proof techniques such as loop invariants would greatly reduce the need of testing. You would prove that each simple method is correct, and then put the proven components together and prove that they work together as they should. Once it is proved that `main` works correctly, no testing is required. Some researchers were so excited about these techniques that they tried to omit the programming step altogether. The designer would write down the program requirements, using the notation of formal logic. An automatic prover would prove that such a program could be written and generate the program as part of its proof.

Unfortunately, in practice these methods never worked very well. The logical notation to describe program behavior is complex. Even simple scenarios require many formulas. It is easy enough to express the idea that a method is supposed to compute a^n, but the logical formulas describing all methods in a program that controls an airplane, for instance, would fill many pages. These formulas are created by humans, and humans make errors when they deal with difficult and tedious tasks. Experiments showed that instead of buggy programs, programmers wrote buggy logic specifications and buggy program proofs.

Van der Linden [2, p. 287], gives some examples of complicated proofs that are much harder to verify than the programs they are trying to prove.

Program proof techniques are valuable for proving the correctness of individual methods that make computations in nonobvious ways. At this time, though, there is no hope to prove any but the most trivial programs correct in such a way that the specification and the proof can be trusted more than the program. There is hope that correctness proofs will become more applicable to real-life programming situations in the future. However, engineering and management are at least as important as mathematics and logic for the successful completion of large software projects.

6.6 Using a Debugger

As you have undoubtedly realized by now, computer programs rarely run perfectly the first time. At times, it can be quite frustrating to find the bugs. Of course, you can insert print commands, run the program, and try to analyze the printout. If the printout does not clearly point to the problem, you may need to add and remove print commands and run the program again. That can be a time-consuming process.

Modern development environments contain special programs, called debuggers, that help you locate bugs by letting you follow the execution of a program. You can stop and restart your program and see the contents of variables whenever your program is temporarily stopped. At each stop, you have the choice of what variables to inspect and how many program steps to run until the next stop.

A debugger is a program that you can use to execute another program and analyze its run-time behavior.

Some people feel that debuggers are just a tool to make programmers lazy. Admittedly some people write sloppy programs and then fix them up with a debugger, but the majority of programmers make an honest effort to write the best program they can before trying to run it through a debugger. These programmers realize that a debugger, while more convenient than print commands, is not cost-free. It does take time to set up and carry out an effective debugging session.

In actual practice, you cannot avoid using a debugger. The larger your programs get, the harder it is to debug them simply by inserting print commands. You will find that the time investment to learn about a debugger is amply repaid in your programming career.

Like compilers, debuggers vary widely from one system to another. On some systems they are quite primitive and require you to memorize a small set of arcane commands; on others they have an intuitive window interface. The screen shots in this chapter show the debugger in the Eclipse development environment, downloadable for free from the Eclipse Foundation web site [4]. Other integrated environments, such as BlueJ, also include debuggers. A free standalone debugger called JSwat is available from the JSwat Graphical Java Debugger web page [5].

You will have to find out how to prepare a program for debugging and how to start a debugger on your system. If you use an integrated development environment, which contains an editor, compiler, and debugger, this step is usually very easy. You just build the program in the usual way and pick a menu command to start debugging. On some systems, you must manually build a debug version of your program and invoke the debugger.

Once you have started the debugger, you can go a long way with just three debugging commands: “set breakpoint”, “single step”, and “inspect variable”. The names and keystrokes or mouse clicks for these commands differ widely between debuggers, but all debuggers support these basic commands. You can find out how, either from the documentation or a lab manual, or by asking someone who has used the debugger before.

You can make effective use of a debugger by mastering just three concepts: breakpoints, single-stepping, and inspecting variables.

264

265

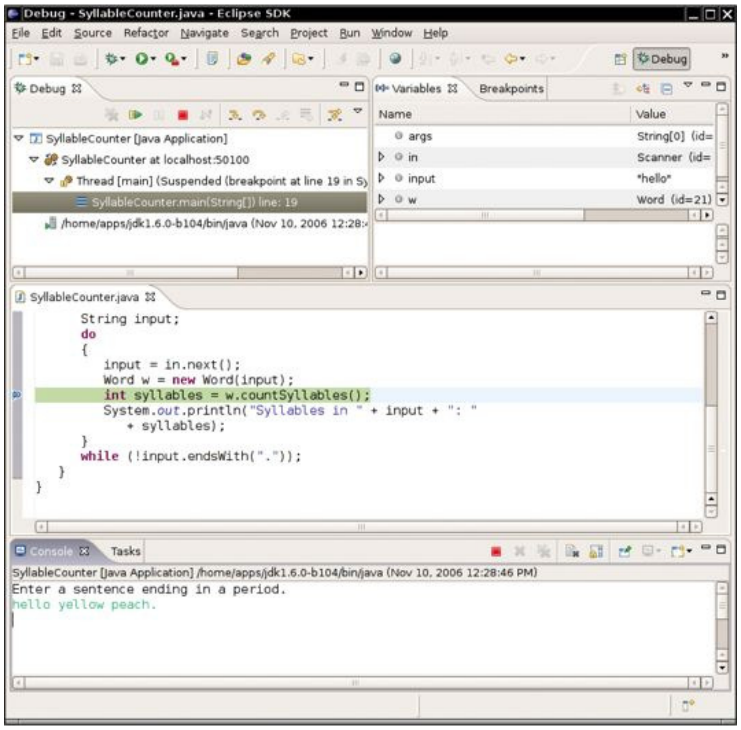
When you start the debugger, it runs at full speed until it reaches a *breakpoint*. Then execution stops, and the breakpoint that causes the stop is displayed (see Figure 5). You can now inspect variables and step through the program a line at a time, or continue running the program at full speed until it reaches the next breakpoint. When the program terminates, the debugger stops as well.

When a debugger executes a program, the execution is suspended whenever a breakpoint is reached.

Breakpoints stay active until you remove them, so you should periodically clear the breakpoints that you no longer need.

Once the program has stopped, you can look at the current values of variables. Again, the method for selecting the variables differs among debuggers. Some debuggers always show you a window with the current local variables. On other debuggers you issue a command such as "inspect variable" and type in or click on the variable. The 265
debugger then displays the contents of the variable. If all variables contain what you 266
expected, you can run the program until the next point where you want to stop.

Figure 5


Debug - SyllableCounter.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Debug
Variables
Breakpoints
SyllableCounter.java
Console
Tasks

```
SyllableCounter [Java Application]
   SyllableCounter at localhost:50100
      Thread [main] (Suspended (breakpoint at line 19 in Sy
         SyllableCounter.main(String[]) line: 19
   /home/apps/jdk1.6.0-b104/bin/java (Nov 10, 2006 12:28:
```

Name	Value
args	String[0] (id=
in	Scanner (id=
input	"hello"
w	Word (id=21)

```java
      String input;
      do
      {
         input = in.next();
         Word w = new Word(input);
         int syllables = w.countSyllables();
         System.out.println("Syllables in " + input + ": "
            + syllables);
      }
      while (!input.endsWith("."));
   }
}
```

```
SyllableCounter [Java Application] /home/apps/jdk1.6.0-b104/bin/java (Nov 10, 2006 12:28:46 PM)
Enter a sentence ending in a period.
hello yellow peach.
```

Stopping at a Breakpoint

When inspecting objects, you often need to give a command to "open up" the object, for example by clicking on a tree node. Once the object is opened up, you see its instance variables (see Figure 6).

Running to a breakpoint gets you there speedily, but you don't know how the program got there. You can also step through the program a line at a time. Then you know how the program flows, but it can take a long time to step through it. The *single-step command* executes the current line and stops at the next program line. Most debuggers have two single-step commands, one called *step into*, which steps inside method calls, and one called *step over*, which skips over method calls.

The single-step command executes the program one line at a time.

For example, suppose the current line is

```
String input = in.next();
Word w = new Word(input);
int syllables = w.countSyllables();
System.out.println("Syllables in " + input + ": " +
syllables);
```

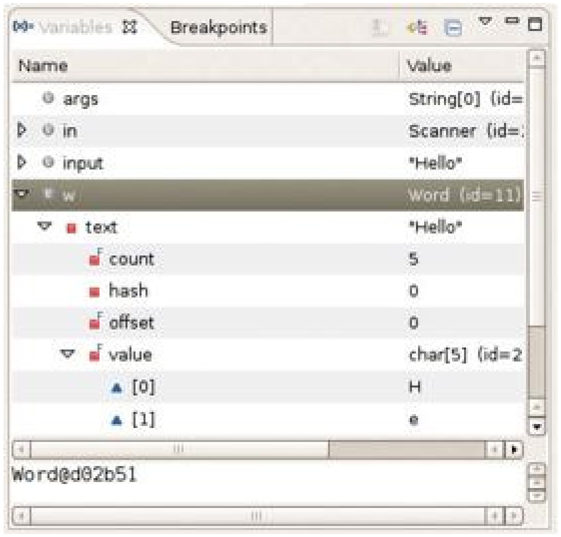
When you step over method calls, you get to the next line:

```
String input = in.next();
Word w = new Word(input);
int syllables = w.countSyllables();
System.out.println("Syllables in " + input + ": " +
syllables);
```

However, if you step into method calls, you enter the first line of the `countSyllables` method.

```
public int countSyllables()
{
      int count = 0;
      int end = text.length() - 1;
      . . .
}
```

Figure 6



Name	Value
args	String[0] (id=
in	Scanner (id=
input	"Hello"
w	Word (id=11)
text	"Hello"
count	5
hash	0
offset	0
value	char[5] (id=2
[0]	H
[1]	e

```
Word@d02b51
```

Inspecting Variables

266

267

You should step *into* a method to check whether it carries out its job correctly. You should step *over* a method if you know it works correctly.

Finally, when the program has finished running, the debug session is also finished. To run the program again, you may be able to reset the debugger, or you may need to exit the debugging program and start over. Details depend on the particular debugger.

SELF CHECK

11. In the debugger, you are reaching a call to `System.out.println`. Should you step into the method or step over it?

12. In the debugger, you are reaching the beginning of a method with a couple of loops inside. You want to find out the return value that is computed at the end of the method. Should you set a breakpoint, or should you step through the method?

6.7 A Sample Debugging Session

To have a realistic example for running a debugger, we will study a `Word` class whose primary purpose is to count the number of syllables in a word. The class uses this rule for counting syllables:

Each group of adjacent vowels (a, e, i, o, u, y) counts as one syllable (for example, the “ea” in “peach” contributes one syllable, but the “e . . . o” in “yellow” counts as two syllables). However, an “e” at the end of a word doesn't count as a syllable. Each word has at least one syllable, even if the previous rules give a count of 0.

Also, when you construct a word from a string, any characters at the beginning or end of the string that aren't letters are stripped off. That is useful when you read the input using the `next` method of the `Scanner` class. Input strings can still contain quotation marks and punctuation marks, and we don't want them as part of the word.

Here is the source code. There are a couple of bugs in this class.

ch06/debugger/Word.java

```
public class Word
{
    /**
      Constructs a word by removing leading and trailing non-
      letter characters, such as punctuation marks.
            @param s the input string
    */
    public Word(String s)
      {
         int i = 0;
         while (i < s.length() &&
!Character.isLetter(s.charAt(i)))
               i++;
         int j = s.length() - 1;
         while (j > i &&
!Character.isLetter(s.charAt(j)))
               j--;
         text = s.substring(i, j);
}

```

```
/**
Returns the text of the word, after removal of the
leading and trailing nonletter characters.
    @return the text of the word
*/
public String getText()
{
    return text;
}

/**
Counts the syllables in the word.
    @return the syllable count
*/
public int countSyllables()
{
    int count = 0;
    int end = text.length() - 1;
    if (end < 0) return 0; // The empty string has no syllables

    // An e at the end of the word doesn't count as a vowel
    char ch = Character.toLowerCase(text.charAt(end));
    if (ch == ‘e’) end--

    boolean insideVowelGroup = false;
    for (int i = 0; i <= end; i++)
    {
        ch = Character.toLowerCase(text.charAt(i));
        if (“aeiouy”.index0f(ch) >= 0)
        {
            // ch is a vowel
            if (!insideVowelGroup)
            {
     // Start of new vowel group
                count++;
                insideVowelGroup = true;
            }
        }
    }
```

```

      // Every word has at least one syllable
      if (count == 0)
         count = 1;
      return count;
   }

   private String text;
}
```

Here is a simple test class. Type in a sentence, and the syllable counts of all words are displayed.

ch06/debugger/SyllableCounter.java

```
import java.util.Scanner;

/**
   This program counts the syllables of all words in a sentence.
*/
public class SyllableCounter
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);

      System.out.println("Enter a sentence ending in a period.");

      String input;
      do
      {
         input = in.next();
         Word w = new Word(input);
         int syllables = w.countSyllables();
         System.out.println("Syllables in " + input + ":"
               + syllables);
      }
      while (!input.endsWith("."));
   }
}
```

Supply this input:

```
hello yellow peach.
```

Then the output is

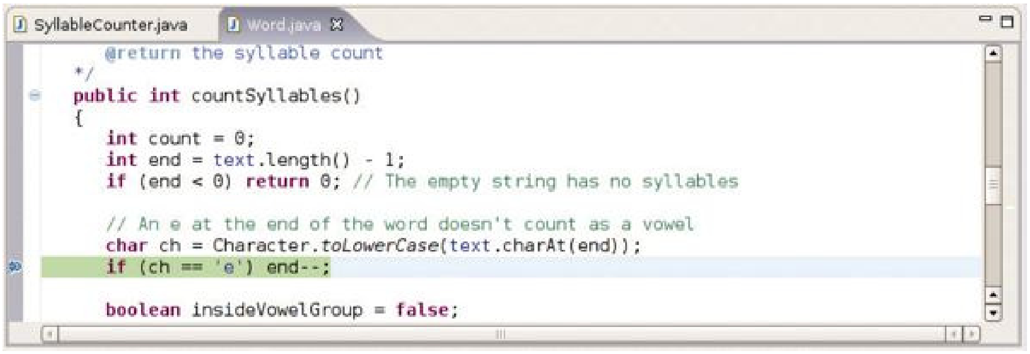
```
Syllables in hello: 1
Syllables in yellow: 1
Syllables in peach.: 1
```

That is not very promising.

First, set a breakpoint in the first line of the `countSyllables` method of the `Word` class, in line 33 of `Word.java`. Then start the program. The program will prompt you for the input. The program will stop at the breakpoint you just set. 269

270

Figure 7



```
    @return the syllable count
  */
public int countSyllables()
{
    int count = 0;
    int end = text.length() - 1;
    if (end < 0) return 0; // The empty string has no syllables

    // An e at the end of the word doesn't count as a vowel
    char ch = Character.toLowerCase(text.charAt(end));
    if (ch == 'e') end--;

    boolean insideVowelGroup = false;
```

Debugging the `countSyllables` Method

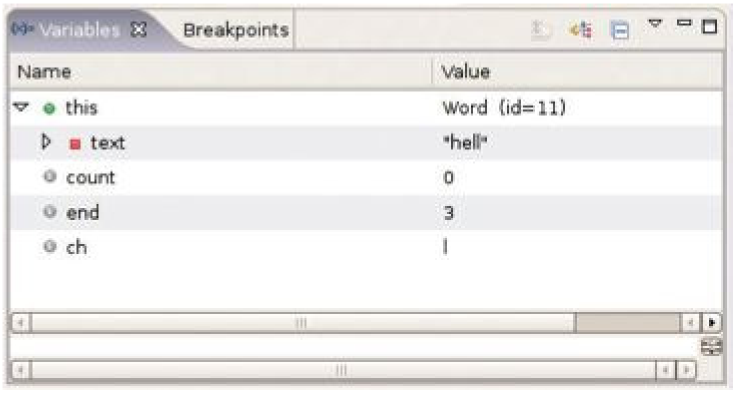
First, the `countSyllables` method checks the last character of the word to see if it is a letter `'e'`. Let's just verify that this works correctly. Run the program to line 41 (see Figure 7).

Now inspect the variable `ch`. This particular debugger has a handy display of all current local and instance variables—see Figure 8. If yours doesn't, you may need to inspect `ch` manually. You can see that `ch` contains the value `'l'`. That is strange. Look at the source code. The `end` variable was set to `text.length() - 1`, the last position in the `text` string, and `ch` is the character at that position.

Looking further, you will find that `end` is set to 3, not 4, as you would expect. And `text` contains the string `"hell"`, not `"hello"`. Thus, it is no wonder that `countSyllables` returns the answer 1. We'll need to look elsewhere for the culprit. Apparently, the `Word` constructor contains an error.

Unfortunately, a debugger cannot go back in time. Thus, you must stop the debugger, set a breakpoint in the `Word` constructor, and restart the debugger. Supply the input once again. The debugger will stop at the beginning of the `Word` constructor. The constructor sets two variables `i` and `j`, skipping past any nonletters at the beginning and the end of the input string. Set a breakpoint past the end of the second loop (see Figure 9) so that you can inspect the values of `i` and `j`.

Figure 8

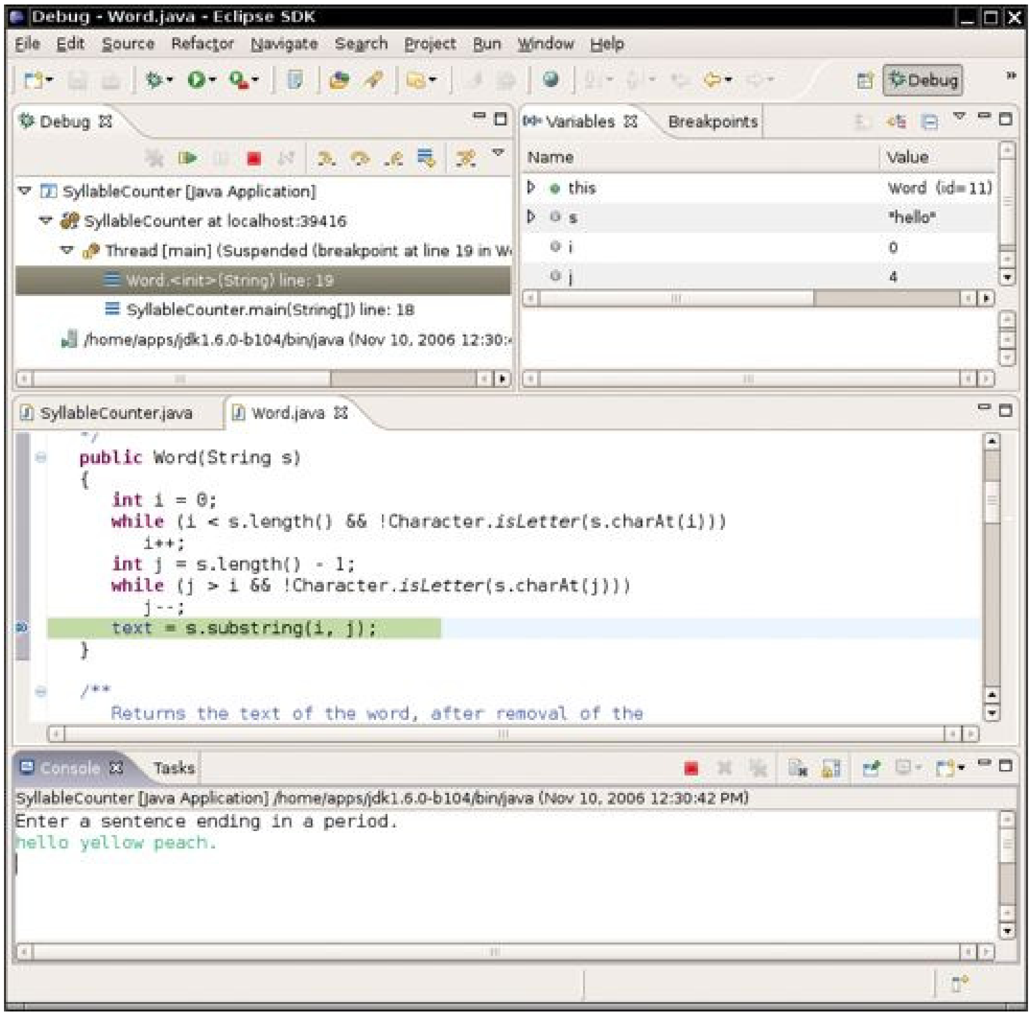


Name	Value
this	Word (id=11)
text	"hell"
count	0
end	3
ch	l

The Current Values of the Local and Instance Variables

270

Figure 9


Debug - Word.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Variables
Breakpoints
SyllableCounter.java
Word.java
Console
Tasks

```
SyllableCounter [Java Application]
  SyllableCounter at localhost:39416
    Thread [main] (Suspended (breakpoint at line 19 in W
      Word.<init>(String) line: 19
      SyllableCounter.main(String[]) line: 18
  /home/apps/jdk1.6.0-b104/bin/java (Nov 10, 2006 12:30:
```

```
Name        Value
this        Word (id=11)
s           "hello"
i           0
j           4
```

```
   */
   public Word(String s)
   {
      int i = 0;
      while (i < s.length() && !Character.isLetter(s.charAt(i)))
         i++;
      int j = s.length() - 1;
      while (j > i && !Character.isLetter(s.charAt(j)))
         j--;
      text = s.substring(i, j);
   }

   /**
      Returns the text of the word, after removal of the
```

```
SyllableCounter [Java Application] /home/apps/jdk1.6.0-b104/bin/java (Nov 10, 2006 12:30:42 PM)
Enter a sentence ending in a period.
hello yellow peach.
```

Debugging the `Word` Constructor

At this point, inspecting `i` and `j` shows that `i` is 0 and `j` is 4. That makes sense—there were no punctuation marks to skip. So why is `text` being set to "`hell`"? Recall that the `substring` method counts positions up to, but not including, the second parameter. Thus, the correct call should be

```
text = s.substring(i, j + 1);
```

This is a very typical off-by-one error.

Fix this error, recompile the program, and try the three test cases again. You will now get the output

```
Syllables in hello: 1
Syllables in yellow: 1
Syllables in peach.: 1
```

As you can see, there still is a problem. Erase all breakpoints and set a breakpoint in the `countSyllables` method. Start the debugger and supply the input `"hello."`. 271

272

When the debugger stops at the breakpoint, start single stepping through the lines of the method. Here is the code of the loop that counts the syllables:

```
boolean insideVowelGroup = false;
for (int i = 0; i <= end; i++)
{
    ch = Character.toLowerCase(text.charAt(i));
    if ("aeiouy".indexOf(ch) >= 0)
    {
        // ch is a vowel
        if (!insideVowelGroup)
        {
              // Start of new vowel group
              count++;
              insideVowelGroup = true;
        }
    }
}
```

In the first iteration through the loop, the debugger skips the `if` statement. That makes sense, because the first letter, `'h'`, isn't a vowel. In the second iteration, the debugger enters the `if` statement, as it should, because the second letter, `'e'`, is a vowel. The `insideVowelGroup` variable is set to `true`, and the vowel counter is incremented. In the third iteration, the `if` statement is again skipped, because the letter `'l'` is not a vowel. But in the fifth iteration, something weird happens. The letter `'o'` is a vowel, and the `if` statement is entered. But the second `if` statement is skipped, and `count` is not incremented again.

Why? The `insideVowelGroup` variable is still true, even though the first vowel group was finished when the consonant `'l'` was encountered. Reading a consonant should set `insideVowelGroup` back to `false`. This is a more subtle logic error, but not an uncommon one when designing a loop that keeps track of the processing state. To fix it, stop the debugger and add the following clause:

```
if ("aeiouy".indexOf(ch) >= 0)
{
      . . .
}
else insideVowelGroup = false;
```

Now recompile and run the test once again. The output is:

A debugger can be used only to analyze the presence of bugs, not to show that a program is bug-free.

```
Syllables in hello: 2
Syllables in yellow: 2
Syllables in peach.: 1
```

Is the program now free from bugs? That is not a question the debugger can answer. Remember: Testing can show only the presence of bugs, not their absence.

SELF CHECK

13. What caused the first error that was found in this debugging session?

14. What caused the second error? How was it detected?

HOW TO 6.2: Debugging

Now you know about the mechanics of debugging, but all that knowledge may still leave you helpless when you fire up a debugger to look at a sick program. There are a number of strategies that you can use to recognize bugs and their causes.

Step 1 Reproduce the error.

As you test your program, you notice that your program sometimes does something wrong. It gives the wrong output, it seems to print something completely random, it goes in an infinite loop, or it crashes. Find out exactly how to reproduce that behavior. What numbers did you enter? Where did you click with the mouse?

Run the program again; type in exactly the same answers, and click with the mouse on the same spots (or as close as you can get). Does the program exhibit the same behavior? If so, then it makes sense to fire up a debugger to study this particular problem. Debuggers are good for analyzing particular failures. They aren't terribly useful for studying a program in general.

Step 2 Simplify the error.

Before you fire up a debugger, it makes sense to spend a few minutes trying to come up with a simpler input that also produces an error. Can you use shorter words or simpler numbers and still have the program misbehave? If so, use those values during your debugging session.

Step 3 Divide and conquer.

Now that you have a particular failure, you want to get as close to the failure as possible. The key point of debugging is to locate the code that produces the failure. Just as with real insect pests, finding the bug can be hard, but once you find it, squashing it is usually the easy part. Suppose your program dies with a division by 0. Because there are many division operations in a typical program, it is often not feasible to set breakpoints to all of them. Instead, use a technique of divide and conquer. Step over the methods in `main`, but don't step inside them. Eventually, the failure will happen again. Now you know which method contains the bug: It is the last method that was called from `main` before the program died. Restart the debugger and go back to that line in `main`, then step inside that method. Repeat the process.

Use the divide-and-conquer technique to locate the point of failure of a program.

Eventually, you will have pinpointed the line that contains the bad division. Maybe it is completely obvious from the code why the denominator is not correct. If not, you need to find the location where it is computed. Unfortunately, you can't go back in the debugger. You need to restart the program and move to the point where the denominator computation happens.

Step 4 Know what your program should do.

During debugging, compare the actual contents of variables against the values you know they should have.

A debugger shows you what the program does. You must know what the program *should* do, or you will not be able to find bugs. Before you trace through a loop, ask yourself how many iterations you expect the program to make. Before you inspect a variable, ask yourself what you expect to see. If you have no clue, set 273
aside some time and think first. Have a calculator handy to make independent 274
computations. When you know what the value should be, inspect the variable. This is the moment of truth. If the program is still on the right track, then that value is what you expected, and you must look further for the bug. If the value is different, you may be on to something. Double-check your computation. If you are sure your value is correct, find out why your program comes up with a different value.

In many cases, program bugs are the result of simple errors such as loop termination conditions that are off by one. Quite often, however, programs make computational errors. Maybe they are supposed to add two numbers, but by accident the code was written to subtract them. Unlike your calculus instructor, programs don't make a special effort to ensure that everything is a simple integer (and neither do real-world problems). You will need to make some calculations with large integers or nasty floating-point numbers. Sometimes these calculations can be avoided if you just ask yourself, “Should this quantity be positive? Should it be larger than that value?” Then inspect variables to verify those theories.

Step 5 Look at all details.

When you debug a program, you often have a theory about what the problem is. Nevertheless, keep an open mind and look around at all details. What strange messages are displayed? Why does the program take another unexpected action? These details count. When you run a debugging session, you really are a detective who needs to look at every clue available.

If you notice another failure on the way to the problem that you are about to pin down, don't just say, “I'll come back to it later”. That very failure may be the original cause for your current problem. It is better to make a note of the current problem, fix what you just found, and then return to the original mission.

Step 6 Make sure you understand each bug before you fix it.

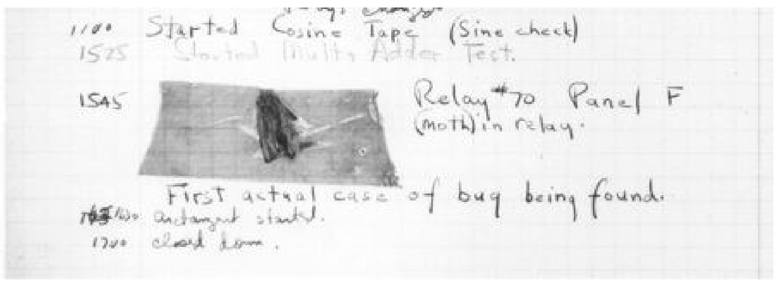
Once you find that a loop makes too many iterations, it is very tempting to apply a “Band-Aid” solution and subtract 1 from a variable so that the particular problem doesn't appear again. Such a quick fix has an overwhelming probability of creating trouble elsewhere. You really need to have a thorough understanding of how the program should be written before you apply a fix.

It does occasionally happen that you find bug after bug and apply fix after fix, and the problem just moves around. That usually is a symptom of a larger problem with the program logic. There is little you can do with the debugger. You must rethink the program design and reorganize it.

RANDOM FACT 6.3: The First Bug

According to legend, the first bug was one found in 1947 in the Mark II, a huge electro-mechanical computer at Harvard University. It really was caused by a bug—a moth was trapped in a relay switch. Actually, from the note that the operator left in the log book next to the moth (see The First Bug figure), it appears as if the term “bug” had already been in active use at the time.

The pioneering computer scientist Maurice Wilkes wrote: “Somehow, at the Moore School and afterwards, one had always assumed there would be no 274
particular difficulty in getting programs right. I can remember the exact instant in 275
time at which it dawned on me that a great part of my future life would be spent finding mistakes in my own programs.”



1100 Started Cosine Tape (Sine check)
1525 Started Multi Adder Test.
1545 Relay #70 Panel F (moth) in relay.
First actual case of bug being found.
1630 Antangent started.
1700 closed down.

The First Bug

CHAPTER SUMMARY

1. A `while` statement executes a block of code repeatedly. A condition controls how often the loop is executed.
2. An off-by-one error is a common error when programming loops. Think through simple test cases to avoid this type of error.
3. You use a `for` loop when a variable runs from a starting to an ending value with a constant increment or decrement.
4. Loops can be nested. A typical example of nested loops is printing a table with rows and columns.
5. Sometimes, the termination condition of a loop can only be evaluated in the middle of a loop. You can introduce a Boolean variable to control such a loop.
6. Make a choice between symmetric and asymmetric loop bounds.
7. Count the number of iterations to check that your `for` loop is correct.
8. In a simulation, you repeatedly generate random numbers and use them to simulate an activity.
9. A debugger is a program that you can use to execute another program and analyze its run-time behavior.
10. You can make effective use of a debugger by mastering just three concepts: breakpoints, single-stepping, and inspecting variables.
11. When a debugger executes a program, the execution is suspended whenever a breakpoint is reached. 275

276

12. The single-step command executes the program one line at a time.
13. A debugger can be used only to analyze the presence of bugs, not to show that a program is bug-free.
14. Use the divide-and-conquer technique to locate the point of failure of a program.

15. During debugging, compare the actual contents of variables against the values you know they should have.

FURTHER READING

1. E. W. Dijkstra, "Goto Statements Considered Harmful", *Communications of the ACM*, vol. 11, no. 3 (March 1968), pp. 147–148.

2. Peter van der Linden, *Expert C Programming*, Prentice-Hall, 1994.

3. Jon Bentley, *Programming Pearls*, Chapter 4, "Writing Correct Programs", Addison-Wesley, 1986.

4. http://eclipse.org The Eclipse Foundation web site.

5. http://www.bluemarsh.com/java/jswat The JSwat Graphical Java Debugger web page.

6. Kai Lai Chung, *Elementary Probability Theory with Stochastic Processes*, Undergraduate Texts in Mathematics, Springer-Verlag, 1974.

7. Rudolf Flesch, *How to Write Plain English*, Barnes & Noble Books, 1979.

CLASSES, OBJECTS, AND METHODS INTRODUCED IN THIS CHAPTER

```
java.util.Random
     nextDouble
     nextInt
```

REVIEW EXERCISES

★★ **Exercise R6.1.** Which loop statements does Java support? Give simple rules when to use each loop type.

★★ **Exercise R6.2.** What does the following code print?

```
for (int i = 0; i < 10; i++)
{
    for (int j = 0; j < 10; j++)
          System.out.print(i * j % 10);
    System.out.println();
}
```

★★ **Exercise R6.3.** How often do the following loops execute? Assume that `i` is an integer variable that is not changed in the loop body.

a. `for (i = 1; i <= 10; i++) …`

b. `for (i = 0; i < 10; i++) …`

c. `for (i = 10; i > 0; i--) …`

d. `for (i = -10; i <= 10; i++) …`

e. `for (i = 10; i > = 0; i++) …`

f. `for (i = -10; i <= 10; i = i + 2) …`

g. `for (i = -10; i <= 10; i = i + 3) …`

★ **Exercise R6.4.** Rewrite the following `for` loop into a `while` loop.

```
int s = 0;
for (int i = 1; i <= 10; i++) s = s + i;
```

★ **Exercise R6.5.** Rewrite the following `do` loop into a `while` loop.

```
int n = 1;
double x = 0;
double s;
do
{
      s = 1.0 / (n * n);
      x = x + s;
      n++;
}
while (s > 0.01);
```

★ **Exercise R6.6.** What is an infinite loop? On your computer, how can you terminate a program that executes an infinite loop?

★★★ **Exercise R6.7** Give three strategies to implement the following "loop and a half":

```
loop
{
    Read name of bridge
    If not OK, exit loop
    Read length of bridge in feet
    If not OK, exit loop
    Convert length to meters
    Print bridge data
}
```

Use a Boolean variable, a `break` statement, and a method with multiple `return` statements. Which of these three approaches do you find clearest?

★ **Exercise R6.8** Implement a loop that prompts a user to enter a number between 1 and 10, giving three tries to get it right.

★ **Exercise R6.9** Sometimes students write programs with instructions such as "Enter data, 0 to quit" and that exit the data entry loop when the user enters the number 0. Explain why that is usually a poor idea. 277

278

★ **Exercise R6.10.** How would you use a random number generator to simulate the drawing of a playing card?

★ **Exercise R6.11.** What is an "off-by-one error"? Give an example from your own programming experience.

★★ **Exercise R6.12.** Give an example of a `for` loop in which symmetric bounds are more natural. Give an example of a `for` loop in which asymmetric bounds are more natural.

★ **Exercise R6.13** What are nested loops? Give an example where a nested loop is typically used.

★T **Exercise R6.14** Explain the differences between these debugger operations:

- Stepping into a method
- Stepping over a method

★★T **Exercise R6.15** Explain in detail how to inspect the string stored in a `String` object in your debugger.

★★T **Exercise R6.16** Explain in detail how to inspect the information stored in a `Rectangle` object in your debugger.

★★T **Exercise R6.17** Explain in detail how to use your debugger to inspect the balance stored in a `BankAccount` object.

★★T **Exercise R6.18** Explain the divide-and-conquer strategy to get close to a bug in a debugger.

Additional review exercises are available in Wiley PLUS.

PROGRAMMING EXERCISES

★ **Exercise P6.1** *Currency conversion*. Write a program `CurrencyConverter` that asks the user to enter today's exchange rate between U.S. dollars and the euro. Then the program reads U.S. dollar values and converts each to euro values. Stop when the user enters `Q`.

★★★ **Exercise P6.2** *Projectile flight*. Suppose a cannonball is propelled vertically into the air with a starting velocity v_0. Any calculus book will tell us that the position of the ball after t seconds is $s(t) = -0.5 \cdot g \cdot t^2 + v_0 \cdot t$, where g 9.81 m/sec^2 is the gravitational force of the earth. No calculus book ever mentions why someone would want to carry out such an obviously dangerous experiment, so we will do it in the safety of the computer.

In fact, we will confirm the theorem from calculus by a simulation. In our simulation, we will consider how the ball moves in very short time 278
intervals Δt. In a short time interval the velocity v is nearly constant, and 279
we can compute the distance the ball moves as $\Delta s = v \cdot \Delta t$. In our program, we will simply set

```
double deltaT = 0.01;
```

and update the position by

```
s = s + v * deltaT;
```

The velocity changes constantly—in fact, it is reduced by the gravitational force of the earth. In a short time interval, v decreases by $g \cdot \Delta t$, and we must keep the velocity updated as

```
v = v - g * deltaT;
```

In the next iteration the new velocity is used to update the distance.

Now run the simulation until the cannonball falls back to the earth. Get the initial velocity as an input (100 m/sec is a good value). Update the position and velocity 100 times per second, but only print out the position every full second. Also print out the values from the exact formula $s(t) = -0.5 \cdot g \cdot t^2 + v_0 \cdot t$ for comparison. Use a class `Cannonball`.

What is the benefit of this kind of simulation when an exact formula is available? Well, the formula from the calculus book is *not* exact. Actually, the gravitational force diminishes the farther the cannonball is away from the surface of the earth. This complicates the algebra sufficiently that it is not possible to give an exact formula for the actual motion, but the computer simulation can simply be extended to apply a variable gravitational force. For cannonballs, the calculus-book formula is actually good enough, but computers are necessary to compute accurate trajectories for higher-flying objects such as ballistic missiles.

★★ **Exercise P6.3.** Write a program that prints the powers of ten

```
1.0
10.0
100.0
1000.0
10000.0
100000.0
1.0E7
1.0E8
1.0E9
1.0E10
1.0E11
```

Implement a class

```
public class PowerGenerator
{
      /**
     Constructs a power generator.
            @param aFactor the number that will be multiplied
by itself
      */
      public PowerGenerator(int aFactor) { . . . }
```

```
      /**
     Computes the next power.
      */
      public double nextPower() { . . . }
      . . .
}
```

Then supply a test class `PowerGeneratorRunner` that calls `System.out.println(myGenerator.nextPower())` twelve times.

★★ **Exercise P6.4.** The *Fibonacci sequence* is defined by the following rule. The first two values in the sequence are 1 and 1. Every subsequent value is the sum of the two values preceding it. For example, the third value is 1 + 1 = 2, the fourth value is 1 + 2 = 3, and the fifth is 2 + 3 = 5. If f_n denotes the first *n*th value in the Fibonacci sequence, then

$$f_1 = 1$$

$$f_2 = 1$$

$$f_n = f_{n-1} + f_{n-2} \quad \text{if} \quad n > 2$$

Write a program that prompts the user for *n* and prints the *n*th value in the Fibonacci sequence. Use a class `FibonacciGenerator` with a method `nextNumber`.

Hint: There is no need to store all values for f_n. You only need the last two values to compute the next one in the series:

```
fold1 = 1;
fold2 = 1;
```

```
fnew = fold1 + fold2;
```

After that, discard `fold2`, which is no longer needed, and set `fold2` to `fold1` and `fold1` to `fnew`.

Your generator class will be tested with this runner program:

```
public class FibonacciRunner
{
      public static void main(String[] args)
      {
            Scanner in = new Scanner(System.in);

            System.out.println("Enter n:");
            int n = in.nextInt();
            FibonacciGenerator fg = new
FibonacciGenerator();
            for (int i = 1; i <= n; i++)
               System.out.println(fg.nextNumber());
      }
}
```

★★ **Exercise P6.5.** *Mean and standard deviation*. Write a program that reads a set of floating-point data values from the input. When the user indicates the 280 end of input, print out the count of the values, the average, and the standard 281 deviation. The average of a data set $x_1, \ldots, x_n$ is

$$\overline{x} = \frac{\Sigma x_i}{n}$$

where $\sum x_i = x_1 + \ldots + x_n$ is the sum of the input values. The standard deviation is

$$s = \sqrt{\frac{\Sigma (x_i - \overline{x})^2}{n - 1}}$$

However, that formula is not suitable for our task. By the time you have computed the mean, the individual x_i are long gone. Until you know how to save these values, use the numerically less stable formula

$$s = \sqrt{\frac{\Sigma x_i^2 - \frac{1}{n}\left(\Sigma x_i\right)^2}{n - 1}}$$

You can compute this quantity by keeping track of the count, the sum, and the sum of squares in the `DataSet` class as you process the input values.

★★ **Exercise P6.6.** *Factoring of integers*. Write a program that asks the user for an integer and then prints out all its factors in increasing order. For example, when the user enters 150, the program should print

```
2
3
5
5
```

Use a class `FactorGenerator` with a constructor `FactorGenerator(int numberToFactor)` and methods `nextFactor` and `hasMoreFactors`. Supply a class `FactorPrinter` whose `main` method reads a user input, constructs a `FactorGenerator` object, and prints the factors.

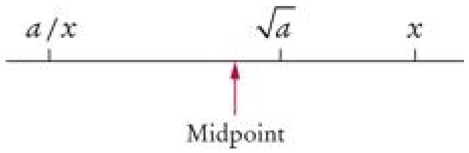
★★ **Exercise P6.7.** *Prime numbers*. Write a program that prompts the user for an integer and then prints out all prime numbers up to that integer. For example, when the user enters 20, the program should print

```
2
3
5
7
11
13
17
19
```

Recall that a number is a prime number if it is not divisible by any number except 1 and itself.

Supply a class `PrimeGenerator` with a method `nextPrime`.

★★ **Exercise P6.8.** The *Heron method* is a method for computing square roots that was known to the ancient Greeks. If x is a guess for the value $\sqrt{a}$, then the average of x and a/x is a better guess.



Implement a class `RootApproximator` that starts with an initial guess of 1 and whose `nextGuess` method produces a sequence of increasingly better guesses. Supply a method `hasMoreGuesses` that returns `false` if two successive guesses are sufficiently close to each other (that is, they differ by no more than a small value ε). Then test your class like this:

```
RootApproximator approx = new RootApproximator(a,
epsilon);
while (approx.hasMoreGuesses())
      System.out.println(approx.nextGuess());
```

★★ **Exercise P6.9.** The best known iterative method for computing the roots of a function f (that is, the x-values for which $f(x)$ is 0) is Newton–Raphson approximation. To find the zero of a function whose derivative is also known, compute

$$x_{\text{new}} = x_{\text{old}} - f\left(x_{\text{old}}\right) / f'\left(x_{\text{old}}\right) .$$

For this exercise, write a program to compute nth roots of floating-point numbers. Prompt the user for a and n, then obtain $\sqrt[n]{a}$ by computing a zero of the function $f(x)\ x^n - a$. Follow the approach of Exercise P6.8.

★★ **Exercise P6.10.** The value of e^X can be computed as the power series

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

where $n!=1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$.

Write a program that computes e^x using this formula. Of course, you can't compute an infinite sum. Just keep adding values until an individual summand (term) is less than a certain threshold. At each step, you need to compute the new term and add it to the total. Update these terms as follows:

```
term = term * x / n;
```

Follow the approach of the preceding two exercises, by implementing a class `ExpApproximator`. Its first guess should be 1.

★ **Exercise P6.11.** Write a program RandomDataAnalyzer that generates 100 random numbers between 0 and 1000 and adds them to a DataSet. Print out the average and the maximum.

★★ **Exercise P6.12.** Program the following simulation: Darts are thrown at random points onto the square with corners (1,1) and (−1,−1). If the dart 282 lands inside the 283 unit circle (that is, the circle with center (0,0) and radius 1), it is a hit. Otherwise it is a miss. Run this simulation and use it to determine an approximate value for π. Extra credit if you explain why this is a better method for estimating π than the Buffon needle program.

★★★G **Exercise P6.13.** *Random walk.* Simulate the wandering of an intoxicated person in a square street grid. Draw a grid of 20 streets horizontally and 20 streets vertically. Represent the simulated drunkard by a dot, placed in the middle of the grid to start. For 100 times, have the simulated drunkard randomly pick a direction (east, west, north, south), move one block in the chosen direction, and draw the dot. (One might expect that on average the person might not get anywhere because the moves to different directions cancel one another out in the long run, but in fact it can be shown with probability 1 that the person eventually moves outside any finite region. See, for example, [6, Chapter 8] for more details.) Use classes for the grid and the drunkard.

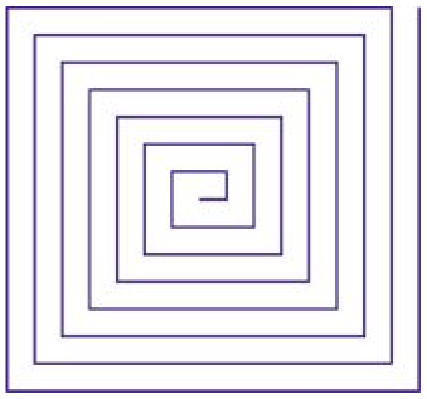
★★★G **Exercise P6.14.** This exercise is a continuation of Exercise P6.2. Most cannonballs are not shot upright but at an angle. If the starting velocity has magnitude v and the starting angle is α, then the velocity is a vector with components $v_x = v \cdot \cos(\alpha)$, $v_y = v \cdot \sin(\alpha)$. In the x-direction the velocity does not change. In the *y*-direction the gravitational force takes its toll. Repeat the simulation from the previous exercise, but update the *x* and *y* components of the location and the velocity separately. In every iteration, plot the location of the cannonball on the graphics display as a tiny circle. Repeat until the cannonball has reached the earth again.

This kind of problem is of historical interest. The first computers were designed to carry out just such ballistic calculations, taking into account the diminishing gravity for high-flying projectiles and wind speeds.

★G **Exercise P6.15.** Write a graphical application that displays a checkerboard with 64 squares, alternating white and black.

★★G **Exercise P6.16.** Write a graphical application that prompts a user to enter a number `n` and that draws `n` circles with random diameter and random location. The circles should be completely contained inside the window.

★★★G **Exercise P6.17.** Write a graphical application that draws a spiral, such as the following:



★★G **Exercise P6.18.** It is easy and fun to draw graphs of curves with the Java graphics library. Simply draw 100 line segments joining the points (x, $f(x)$) and ($x + d$, $f(x + d)$), where x ranges from x_{min} to x_{max} and $d = (x_{\text{max}}$

– x_{max})/100. Draw the curve $f(x) = 0.00005x^3 - 0.03x^2 + 4x + 200$, where x ranges from 0 to 400 in this fashion.

★★★G **Exercise P6.19.** Draw a picture of the “four-leaved rose” whose equation in polar coordinates is $r = \cos(2\theta)$. Let θ go from 0 to 2π in 100 steps. Each time, compute `r` and then compute the (x,y) coordinates from the polar coordinates by using the formula

$$x = r \cos \theta , \quad y = r \sin \theta$$

Additional review exercises are available in Wiley Plus.

PROGRAMMING PROJECTS

★★★ **Project 6.1.** *Flesch Readability Index.* The following index [7] was invented by Flesch as a tool to gauge the legibility of a document without linguistic analysis.

- Count all words in the file. A *word* is any sequence of characters delimited by white space, whether or not it is an actual English word.
- Count all syllables in each word. To make this simple, use the following rules: Each *group* of adjacent vowels (a, e, i, o, u, y) counts as one syllable (for example, the “ea” in “real” contributes one syllable, but the “e … a” in “regal” count as two syllables). However, an “e” at the end of a word doesn't count as a syllable. Also, each word has at least one syllable, even if the previous rules give a count of 0.
- Count all sentences. A sentence is ended by a period, colon, semicolon, question mark, or exclamation mark.
- The index is computed by

$$\begin{aligned}\text{Index} &= 206.835 \\ &- 84.6 \times (\text{Number of syllables} / \text{Number of words}) \\ &- 1.015 \times (\text{Number of words} / \text{Number of sentences})\end{aligned}$$

rounded to the nearest integer.

The purpose of the index is to force authors to rewrite their text until the index is high enough. This is achieved by reducing the length of sentences and by removing long words. For example, the sentence

The following index was invented by Flesch as a simple tool to estimate the legibility of a document without linguistic analysis.

can be rewritten as

Flesch invented an index to check whether a text is easy to read. To compute the index, you need not look at the meaning of the words.

Flesch's book [7] contains delightful examples of translating government regulations into “plain English”.

This index is a number, usually between 0 and 100, indicating how difficult the text is to read. Some example indices for random material from various publications are:

Comics	95
Consumer ads	82
Sports Illustrated	65
Time	57
New York Times	39
Auto insurance policy	10
Internal Revenue Code	– 6

Translated into educational levels, the indices are:

91–100	5th grader
81–90	6th grader
71–80	7th grader
66–70	8th grader
61–65	9th grader
51–60	High school student
31–50	College student
0–30	College graduate
Less than 0	Law school graduate

Your program should read a text file in, compute the legibility index, and print out the equivalent educational level. Use classes `Word` and `Document`.

★★★ **Project 6.2.** *The game of Nim.* This is a well-known game with a number of variants. We will consider the following variant, which has an interesting winning strategy. Two players alternately take marbles from a pile. In each move, a player chooses how many marbles to take. The player must take at least one but at most half of the marbles. Then the other player takes a turn. The player who takes the last marble loses.

Write a program in which the computer plays against a human opponent. Generate a random integer between 10 and 100 to denote the initial size of the pile. Generate a random integer between 0 and 1 to decide whether the computer or the human takes the first turn. Generate a random 285
integer between 0 and 1 to decide whether the computer plays *smart* or 286
stupid. In stupid mode, the computer simply takes a random legal value (between 1 and $n/2$) from the pile whenever it has a turn. In smart mode the computer takes off enough marbles to make the size of the pile a power of two minus 1—that is, 3, 7, 15, 31, or 63. That is always a legal move, except if the size of the pile is currently one less than a power of 2. In that case, the computer makes a random legal move.

Note that the computer cannot be beaten in smart mode when it has the first move, unless the pile size happens to be 15, 31, or 63. Of course, a human player who has the first turn and knows the winning strategy can win against the computer.

Be sure to use classes `Pile`, `Player`, and `Game` in your implementation. A player can be either stupid, smart, or human. (Human `Player` objects prompt for input.)

ANSWERS TO SELF-CHECK QUESTIONS

1. Never
2. The `waitForBalance` method would never return due to an infinite loop
3.

```
int i = 1;
while (i <= n)
{
      double interest = balance * rate / 100;
      balance = balance + interest;
      i++;
}
```

4. 11 times
5. Change the inner loop to `for (int j = 1; j <= width; j++)`
6. 20
7. Because we don't know whether the next input is a number or the letter `Q`
8. No. If *all* input values are negative, the maximum is also negative. However, the `maximum` field is initialized with 0. With this simplification, the maximum would be falsely computed as 0
9. `int n = generator.nextInt(2); //`0 = heads, 1 = tails
10. The program repeatedly calls `Math.toRadians(angle)`. You could simply call `Math.toRadians(180)` to compute π.
11. You should step over it because you are not interested in debugging the internals of the `println` method.
12. You should set a breakpoint. Stepping through loops can be tedious.

13. The programmer misunderstood the second parameter of the substring method—it is the index of the first character not to be included in the substring.

14. The second error was caused by failing to reset `insideVowelGroup` to false at the end of a vowel group. It was detected by tracing through the loop and noticing that the loop didn't enter the conditional statement that increments the vowel count.