CHAPTER GOALS

- To be able to program loops with the `while` and `for` statements
- To avoid infinite loops and off-by-one errors
- To be able to use common loop algorithms
- To understand nested loops
- To implement simulations
- 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.
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 percent 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 percent of $10,000) of interest. The interest gets added to the bank account. Next year, the interest is $525 (5 percent of $10,500), and the balance is $11,025.

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

```java
while (condition) {
    statement
}
```

keeps executing the statement while the condition is true.

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

Figure 1 shows the flow of execution of this loop.

![Figure 1 Execution of a while Loop](image)

For the full text of the sample program that solves our investment problem, see ch06/invest1/Investment.java in your source code, or view it in WileyPLUS.
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 2).

When you declare a variable inside the loop body, the variable is created for each iteration of the loop and removed after the end of each iteration. For example, consider the interest variable in this loop:

```java
while (balance < targetBalance)
{
    years++;
    double interest = balance * rate / 100;
    // A new interest variable is created
    // in each iteration
    balance = balance + interest;
} // interest no longer declared here
```

If a variable needs to be updated in multiple loop iterations, do not declare it inside the loop. For example, it would not make sense to declare the balance variable inside this loop.

**Figure 2** Flowchart of a while Loop

---

**Syntax 6.1 The while Statement**

<table>
<thead>
<tr>
<th>Syntax</th>
<th>while (condition) statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>This variable is declared outside the loop and updated in the loop.</td>
</tr>
<tr>
<td></td>
<td>double balance = 0;</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>while (balance &lt; TARGET)</td>
</tr>
<tr>
<td></td>
<td>{</td>
</tr>
<tr>
<td></td>
<td>double interest = balance * RATE / 100;</td>
</tr>
<tr>
<td></td>
<td>balance = balance + interest;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>These statements are executed while the condition is true.</td>
</tr>
<tr>
<td></td>
<td>Braces are not required if the body contains a single statement.</td>
</tr>
<tr>
<td></td>
<td>Lining up braces is a good idea. See page 154.</td>
</tr>
<tr>
<td></td>
<td>Beware of “off-by-one” errors in the loop condition. See page 200.</td>
</tr>
<tr>
<td></td>
<td>Don’t put a semicolon here! See page 207.</td>
</tr>
<tr>
<td></td>
<td>This variable is created in each loop iteration.</td>
</tr>
<tr>
<td></td>
<td>If the condition never becomes false, an infinite loop occurs. See page 200.</td>
</tr>
</tbody>
</table>
The following loop,

```java
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 Special Topic 6.3 on page 218).

### Table 1: while Loop Examples

<table>
<thead>
<tr>
<th>Loop</th>
<th>Output</th>
<th>Explanation</th>
</tr>
</thead>
</table>
| i = 0; sum = 0; while (sum < 10) {
  i++; sum = sum + i;
  \textbf{Print} \ i \ \textbf{and} \ sum;
} | 1 1 2 3 3 6 4 10 | When sum is 10, the loop condition is false, and the loop ends. |
| i = 0; sum = 0; while (sum < 10) {
  i++; sum = sum - i;
  \textbf{Print} \ i \ \textbf{and} \ sum;
} | 1 -1 2 -3 3 -6 4 -10 . . . | Because sum never reaches 10, this is an “infinite loop” (see Common Error 6.1 on page 200). |
| i = 0; sum = 0; while (sum < 0) {
  i++; sum = sum - i;
  \textbf{Print} \ i \ \textbf{and} \ sum;
} | (No output) | The statement sum < 0 is false when the condition is first checked, and the loop is never executed. |
| i = 0; sum = 0; while (sum >= 10) {
  i++; sum = sum + i;
  \textbf{Print} \ i \ \textbf{and} \ sum;
} | (No output) | The programmer probably thought, “Stop when the sum is at least 10.” However, the loop condition controls when the loop is executed, not when it ends. |
| i = 0; sum = 0; while (sum < 10) ; {
  i++; sum = sum + i;
  \textbf{Print} \ i \ \textbf{and} \ sum;
} | (No output, program does not terminate) | Note the semicolon before the \textbf{. This loop has an empty body. It runs forever, checking whether sum < 10 and doing nothing in the body (see Common Error 6.4 on page 207). |
1. How many times is the following statement in the loop executed?
   \[ \text{while (false) statement;} \]
2. What would happen if \text{RATE} was set to 0 in the main method of the InvestmentRunner program?

**Productivity Hint 6.1**

**Hand-Tracing Loops**

In Programming Tip 5.2, you learned about the method of hand-tracing. This method is particularly effective for understanding how a loop works.

Consider this example loop. What value is displayed?

```java
int n = 1729;
int sum = 0;
while (n > 0) {
    int digit = n % 10;
    sum = sum + digit;
    n = n / 10;
}
System.out.println(sum);
```

1. There are three variables: \(n\), \(sum\), and \(digit\). The first two variables are initialized with 1729 and 0 before the loop is entered.

<table>
<thead>
<tr>
<th>(n)</th>
<th>(sum)</th>
<th>(digit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1729</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. Because \(n\) is positive, enter the loop.

3. The variable \(digit\) is set to 9 (the remainder of dividing 1729 by 10). The variable \(sum\) is set to \(0 + 9 = 9\). Finally, \(n\) becomes 172. (Recall that the remainder in the division 1729 / 10 is discarded because both arguments are integers.). Cross out the old values and write the new ones under the old ones.

<table>
<thead>
<tr>
<th>(n)</th>
<th>(sum)</th>
<th>(digit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1729</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>172</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
4. Because $n > 0$, we repeat the loop. Now digit becomes 2, sum is set to $9 + 2 = 11$, and $n$ is set to 17.

<table>
<thead>
<tr>
<th>n</th>
<th>sum</th>
<th>digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1729</td>
<td>0</td>
<td>172</td>
</tr>
<tr>
<td>172</td>
<td>9</td>
<td>17</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>18</td>
<td>7</td>
</tr>
</tbody>
</table>

5. Because $n$ is still not zero, we repeat the loop, setting digit to 7, sum to $11 + 7 = 18$, and $n$ to 1.

<table>
<thead>
<tr>
<th>n</th>
<th>sum</th>
<th>digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1729</td>
<td>0</td>
<td>172</td>
</tr>
<tr>
<td>172</td>
<td>9</td>
<td>17</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>18</td>
<td>7</td>
</tr>
</tbody>
</table>

6. We enter the loop one last time. Now digit is set to 1, sum to 19, and $n$ becomes zero.

<table>
<thead>
<tr>
<th>n</th>
<th>sum</th>
<th>digit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1729</td>
<td>0</td>
<td>172</td>
</tr>
<tr>
<td>172</td>
<td>9</td>
<td>17</td>
</tr>
<tr>
<td>17</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>18</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>19</td>
<td>1</td>
</tr>
</tbody>
</table>

7. The condition $n > 0$ is now false, and we continue with the output statement after the loop. The value that is output is 19.

Of course, you can get the same answer simply by running the code. The hope is that by hand-tracing, you gain an insight. Consider again what happens in each iteration:

- We extract the last digit of $n$.
- We add that digit to sum.
- We strip the digit off $n$.

In other words, the loop forms the sum of the digits in $n$. You now know what the loop does for any value of $n$, not just the one in the example.

Why would anyone want to form the sum of the digits? Operations of this kind are useful for checking the validity of credit card numbers and other forms of ID number—see Exercise P6.2.
Chapter 6 Iteration

Infinite Loops

One of the most annoying loop errors 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:

```java
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:

```java
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.)

Off-by-One Errors

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

```java
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.

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 percent. 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.

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 percent. 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.

**Special Topic 6.1**

**do Loops**

Special Topic 6.1 discusses the do loop, an optional loop construct that tests the loop condition at the end of the loop body.

### 6.2 for Loops

One of the most common loop types has the form

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

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

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

Available online in WileyPLUS and at www.wiley.com/college/horstmann.
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 Special Topic 6.2).

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

A for loop can be used to find out the size of our $10,000 investment if 5 percent 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.

In our loop, we let $i$ go from 1 to $\text{numberOfYears}$, the number of years for which we want to compound interest.

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

Figure 3 shows the corresponding flowchart. Figure 4 shows the flow of execution. The complete program is on page 205.

---

**Figure 3**
Flowchart of a for Loop
Another common use of the for loop is to traverse all characters of a string:

```java
for (int i = 0; i < str.length(); i++)
{
    char ch = str.charAt(i);
    // Process ch
}
```

Note that the counter variable i starts at 0, and the loop is terminated when i reaches the length of the string. For example, if str has length 5, i takes on the values 0, 1, 2, 3, and 4. These are the valid positions in the string.

Note too that the three slots in the for header can contain any three expressions. You can count down instead of up:

```java
for (int i = 10; i > 0; i--)
```

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

```java
for (int i = -10; i <= 10; i = i + 2)
```

Figure 4  Execution of a for Loop
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.

---

**Table 2** for Loop Examples

<table>
<thead>
<tr>
<th>Loop</th>
<th>Values of i</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i = 0; i &lt;= 5; i++)</td>
<td>0 1 2 3 4 5</td>
<td>Note that the loop is executed 6 times. (See Quality Tip 6.4 on page 209.)</td>
</tr>
<tr>
<td>for (i = 5; i &gt;= 0; i--)</td>
<td>5 4 3 2 1 0</td>
<td>Use -- for decreasing values.</td>
</tr>
<tr>
<td>for (i = 0; i &lt; 9; i = i + 2)</td>
<td>0 2 4 6 8</td>
<td>Use i = i + 2 for a step size of 2.</td>
</tr>
<tr>
<td>for (i = 0; i != 9; i = i + 2)</td>
<td>0 2 4 6 8 10 12 14 ... (infinite loop)</td>
<td>You can use &lt; or &lt;=' instead of != to avoid this problem.</td>
</tr>
<tr>
<td>for (i = 1; i &lt;= 20; i = i * 2)</td>
<td>1 2 4 8 16</td>
<td>You can specify any rule for modifying i, such as doubling it in every step.</td>
</tr>
<tr>
<td>for (i = 0; i &lt; str.length(); i++)</td>
<td>0 1 2 ... until the last valid index of the string str</td>
<td>In the loop body, use the expression str.charAt(i) to get the i-th character.</td>
</tr>
</tbody>
</table>
A class to monitor the growth of an investment that accumulates interest at a fixed annual rate.

```java
public class Investment {
    private double balance;
    private double rate;
    private int years;

    /** 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 numberOfYears the number of years to wait
     */
    public void waitYears(int numberOfYears) {
        for (int i = 1; i <= numberOfYears; i++) {
            double interest = balance * rate / 100;
            balance = balance + interest;
            years = years + n;
        }
    }

    /** Gets the current investment balance.
     * @return the current balance
     */
    public double getBalance() {
    }
}
```
3. Rewrite the for loop in the `waitYears` method as a while loop.

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

```java
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++)
{
    if (balance >= targetBalance)
        i = years;    // Bad style—modifies counter
    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.

---

**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 on page 206; then you could write an investment doubling loop as follows:

```java
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!

```java
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?

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

Of course, this loop is supposed to compute $1 + 2 + \cdots + 10 = 55$. 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.

```java
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

```java
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.

### Quality Tip 6.2

**Don’t Use != to Test the End of a Range**

Here is a loop with a hidden danger:

```java
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:

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

### Special Topic 6.2

**Variables Declared in a for Loop Header**

Special Topic 6.2 shows how to declare multiple variables in a `for` loop header, and it explains that such variables are not defined beyond the loop body.

### Quality Tip 6.3

**Symmetric and Asymmetric Bounds**

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

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

Available online in WileyPLUS and at www.wiley.com/college/horstmann.
The values for \( i \) are bounded by the relation \( 1 \leq i \leq n \). Because there are \( \leq \) comparisons on both bounds, the bounds are called **symmetric**.

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

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

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

It is not a good idea to force symmetry artificially:

```c
for (i = 0; i <= \text{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 \( \leq b \) or \( < b \) as a termination condition?

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

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

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

```c
for (i = 0; i < \text{str.length()}; i++) . . .
```

runs \( \text{str.length()} \) times. That makes perfect sense, because there are \( \text{str.length()} \) characters in a string.

The loop with symmetric bounds,

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

is executed \( b - a + 1 \) times. That “+ 1” is the source of many programming errors. For example,

```c
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 “\( \leq \)” 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

\[
\frac{b - a}{c} \quad \text{for the asymmetric loop}
\]

\[
\frac{b - a}{c + 1} \quad \text{for the symmetric loop}
\]

For example, the loop for \( (i = 10; i <= 40; i += 5) \) executes \((40 - 10)/5 + 1 = 7\) times.
In the following sections, we discuss some of the most common algorithms that are implemented as loops. You can use them as starting points for your loop designs.

### 6.3.1 Computing a Total

Computing the sum of a number of inputs is a very common task. Keep a **running total**: a variable to which you add each input value. Of course, the total should be initialized with 0.

```java
double total = 0;
while (in.hasNextDouble())
{
    double input = in.nextDouble();
    total = total + input;
}
```

### 6.3.2 Counting Matches

You often want to know how many values fulfill a particular condition. For example, you may want to count how many uppercase letters are in a string. Keep a **counter**, a variable that is initialized with 0 and incremented whenever there is a match.

```java
int upperCaseLetters = 0;
for (int i = 0; i < str.length(); i++)
{
    char ch = str.charAt(i);
    if (Character.isUpperCase(ch))
    {
        upperCaseLetters++;
    }
}
```

For example, if `str` is the string "Hello, World!", `upperCaseLetters` is incremented twice (when `i` is 0 and 7).

### 6.3.3 Finding the First Match

When you count the values that fulfill a condition, you need to look at all values. However, if your task is to find a match, then you can stop as soon as the condition is fulfilled.

Here is a loop that finds the first lowercase letter in a string. Because we do not visit all elements in a string, a **while** loop is a better choice than a **for** loop:

```java
boolean found = false;
char ch = '?';
int position = 0;
```
while (!found && position < str.length())
{
    ch = str.charAt(position);
    if (Character.isLowerCase(ch)) { found = true; }
    else { position++; }
}

If a match was found, then found is true, ch is the first matching character, and its index is stored in the variable position. If the loop did not find a match, then found remains false and the loop continues until position reaches str.length().

Note that the variable ch is declared outside the while loop because you may want to use it after the loop has finished.

### 6.3.4 Prompting Until a Match is Found

In the preceding example, we searched a string for a character that matches a condition. You can apply the same process to user input. Suppose you are asking a user to enter a positive value < 100. Keep asking until the user provides a correct input:

```java
boolean valid = false;
double input = 0;
while (!valid)
{
    System.out.print("Please enter a positive value < 100: ");
in = in.nextDouble();
    if (0 < input & input < 100) { valid = true; }
    else { System.out.println("Invalid input."); }
}
```

As in the preceding example, the variable input is declared outside the while loop so that you can use it after the loop has finished.

### 6.3.5 Comparing Adjacent Values

When processing a sequence of values in a loop, you sometimes need to compare a value with the value that just preceded it. For example, suppose you want to check whether a sequence of inputs contains adjacent duplicates such as 1 7 2 9 9 4 9.

Now you face a challenge. Consider the typical loop for reading a value:

```java
double input = 0;
while (in.hasNextDouble())
{
    input = in.nextDouble();
    ...
}
```

How can you compare the current input with the preceding one? At any time, input contains the current input, overwriting the previous one.

The answer is to store the previous input, like this:

```java
double input = 0;
while (in.hasNextDouble())
{
```
One problem remains. When the loop is entered for the first time, there is no previous input value. You can solve this problem with an initial input operation outside the loop:

```java
double input = in.nextDouble();
while (in.hasNextDouble()) {
    double previous = input;
    input = in.nextDouble();
    if (input == previous) { System.out.println("Duplicate input"); }
}
```

One problem remains. When the loop is entered for the first time, there is no previous input value. You can solve this problem with an initial input operation outside the loop:

```java
double input = in.nextDouble();
while (in.hasNextDouble()) {
    double previous = input;
    input = in.nextDouble();
    if (input == previous) { System.out.println("Duplicate input"); }
}
```

### 6.3.6 Processing Input with 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.

```java
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.
6.3 Common Loop Algorithms

```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());
    }
}
```

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.

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

Here is a complete program that reads input and analyzes the data. We separate the input handling from the computation of the data set properties by using two classes, DataAnalyzer and DataSet. The DataAnalyzer class handles the input and adds values to a DataSet object with the add method. It then calls the getAverage method and the getMaximum method to obtain the average and maximum of all added data.
Chapter 6 Iteration

ch06/dataset/DataSet.java

```java
/**
 * Computes information about a set of data values.
 */
public class DataSet {
    private double sum;
    private double maximum;
    private int count;

    /**
     * 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;
    }
}
```

Program Run

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
5. How do you compute the total of all positive inputs?

6. What happens with the algorithm in Section 6.3.5 when no input is provided at all? How can you overcome that problem?

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 variable in the add method to the following statement?

   \[
   \text{if (maximum < x) maximum = x};
   \]

How To 6.1 Writing a Loop

This How To walks you through the process of implementing a loop statement. We will illustrate the steps with the following example problem:

Read twelve temperature values (one for each month), and display the number of the month with the highest temperature. For example, according to http://worldclimate.com, the average maximum temperatures for Death Valley are (in order by month):

\[
18.2 \ 22.6 \ 26.4 \ 31.1 \ 36.6 \ 42.2 \\
45.7 \ 44.5 \ 40.2 \ 33.1 \ 24.2 \ 17.6
\]

In this case, the month with the highest temperature (45.7 degrees Celsius) is July, and the program should display 7.

Step 1 Decide what work must be done inside the loop.

Every loop needs to do some kind of repetitive work, such as

- Reading another item.
- Updating a value (such as a bank balance or total).
- Incrementing a counter.

If you can’t figure out what needs to go inside the loop, start by writing down the steps that you would take if you solved the problem by hand. For example, with the temperature reading problem, you might write

   \[
   \text{Read first value.} \\
   \text{Read second value.} \\
   \text{If second value is higher than the first, set highest temperature to that value, highest month to 2.} \\
   \text{Read next value.} \\
   \text{If value is higher than the first and second, set highest temperature to that value, highest month to 3.} \\
   \text{Read next value.} \\
   \text{If value is higher than the highest temperature seen so far, set highest temperature to that value,} \\
   \text{highest month to current month.} \\
   \]

Now look at these steps and reduce them to a set of uniform actions that can be placed into the loop body. The first action is easy:

   \[
   \text{Read next value.}
   \]

The next action is trickier. In our description, we used tests “higher than the first”, “higher than the first and second”, “higher than the highest temperature seen so far”. We need to settle on one test that works for all iterations. The last formulation is the most general.

Similarly, we must find a general way of setting the highest month. We need a variable that stores the current month, running from 1 to 12. Then we can formulate the second loop action:

   \[
   \text{If value is higher than the highest temperature, set highest temperature to that value,} \\
   \text{highest month to current month.}
   \]
Chapter 6  Iteration

Altogether our loop is

```
Loop
  Read next value.
  If value is higher than the highest temperature, set highest temperature to that value,
  highest month to current month.
  Increment current month.
```

**Step 2**  Specify the loop condition.

What goal do you want to reach in your loop? Typical examples are

- Has a counter reached its final value?
- Have you read the last input value?
- Has a value reached a given threshold?

In our example, we simply want the current month to reach 12.

**Step 3**  Determine the loop type.

We distinguish between two major loop types. A *definite* or *count-controlled* loop is executed a definite number of times. In an *indefinite* or *event-controlled* loop, the number of iterations is not known in advance—the loop is executed until some event happens. A typical example of the latter is a loop that reads data until a sentinel is encountered.

Definite loops can be implemented as *for* statements. When you have an indefinite loop, consider the loop condition. Does it involve values that are only set inside the loop body? In that case, you should choose a *do* loop to ensure that the loop is executed at least once before the loop condition is be evaluated. Otherwise, use a *while* loop.

Sometimes, the condition for terminating a loop changes in the middle of the loop body. In that case, you can use a Boolean variable that specifies when you are ready to leave the loop. Follow this pattern:

```java
boolean done = false;
while (!done)
{
  Do some work
  If all work has been completed
  {
    done = true;
  }
else
  {
    Do more work
  }
}
```

Such a variable is called a *flag*.

In summary,

- If you know in advance how many times a loop is repeated, use a *for* loop.
- If the loop must be executed at least once, use a *do* loop.
- Otherwise, use a *while* loop.

In our example, we read 12 temperature values. Therefore, we choose a *for* loop.

**Step 4**  Set up variables for entering the loop for the first time.

List all variables that are used and updated in the loop, and determine how to initialize them. Commonly, counters are initialized with 0 or 1, totals with 0.
In our example, the variables are

- current month
- highest value
- highest month

We need to be careful how we set up the highest temperature value. We can’t simply set it to 0. After all, our program needs to work with temperature values from Antarctica, all of which may be negative.

A good option is to set the highest temperature value to the first input value. Of course, then we need to remember to only read in another 11 values, with the current month starting at 2.

We also need to initialize the highest month with 1. After all, in an Australian city, we may never find a month that is warmer than January.

**Step 5**  Process the result after the loop has finished.

In many cases, the desired result is simply a variable that was updated in the loop body. For example, in our temperature program, the result is the highest month. Sometimes, the loop computes values that contribute to the final result. For example, suppose you are asked to average the temperatures. Then the loop should compute the sum, not the average. After the loop has completed, you are ready compute the average: divide the sum by the number of inputs.

Here is our complete loop.

```
Read first value; store as highest value.
highest month = 1
for (current month = 2; current month <= 12; current month++)
  Read next value.
  If value is higher than the highest value, set highest value to that value,
  highest month to current month.
```

**Step 6**  Trace the loop with typical examples.

Hand trace your loop code, as described in Productivity Hint 6.1 on page 198. Choose example values that are not too complex—executing the loop 3–5 times is enough to check for the most common errors. Pay special attention when entering the loop for the first and last time.

Sometimes, you want to make a slight modification to make tracing feasible. For example, when hand tracing the investment doubling problem, use an interest rate of 20 percent rather than 5 percent. When hand tracing the temperature loop, use 4 data values, not 12.

Let’s say the data are 22.6 36.6 44.5 24.2. Here is the walkthrough:

<table>
<thead>
<tr>
<th>current month</th>
<th>current value</th>
<th>highest month</th>
<th>highest value</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>36.6</td>
<td>2</td>
<td>36.6</td>
</tr>
<tr>
<td>3</td>
<td>44.5</td>
<td>3</td>
<td>44.5</td>
</tr>
<tr>
<td>4</td>
<td>24.2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The trace demonstrates that highest month and highest value are properly set.
Step 7 Implement the loop in Java.

Here’s the loop for our example. Exercise P6.1 asks you to complete the program.

```java
double highestValue = in.nextDouble();
int highestMonth = 1;
for (int currentMonth = 2; currentMonth <= 12; currentMonth++)
{
    double nextValue = in.nextDouble();
    if (nextValue > highestValue)
    {
        highestValue = nextValue;
        highestMonth = currentMonth;
    }
}
```

Credit Card Processing

This Worked Example uses a loop to remove spaces from a credit card number.

Special Topic 6.3

The “Loop and a Half” Problem

Special Topic 6.3 discusses two alternate strategies for implementing a loop whose termination condition is determined halfway into the loop body.

Special Topic 6.4

The break and continue Statements

Special Topic 6.4 discusses the optional break and continue statements. Neither statement is necessary for implementing loops, but they can occasionally make a complex loop more concise.

6.4 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.

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:

```java
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 $i$.

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

Putting both loops together yields two nested loops:

```java
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;
```

For the full text of the program, see ch06/triangle1/ in your source code, or view it in WileyPLUS.

<table>
<thead>
<tr>
<th>Table 3 Nested Loop Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nested Loops</td>
</tr>
<tr>
<td>--------------</td>
</tr>
</tbody>
</table>
| for (i = 1; i <= 3; i++)
{ for (j = 1; j <= 4; j++) { Print "*" } System.out.println();
} | ****
****
**** |
| Prints 3 rows of 4 asterisks each. |
| for (i = 1; i <= 4; i++)
{ for (j = 1; j <= 3; j++) { Print "*" } System.out.println();
} | ***
***
*** |
| Prints 4 rows of 3 asterisks each. |
### Table 3  Nested Loop Examples, continued

<table>
<thead>
<tr>
<th>Nested Loops</th>
<th>Output</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i = 1; i &lt;= 4; i++) { &lt;br&gt;  for (j = 1; j &lt;= i; j++) { Print &quot;*&quot; }&lt;br&gt; System.out.println(); }</td>
<td>* &lt;br&gt; ** &lt;br&gt; *** &lt;br&gt; ****</td>
<td>Prints 4 rows of lengths 1, 2, 3, and 4.</td>
</tr>
<tr>
<td>for (i = 1; i &lt;= 3; i++) { &lt;br&gt;  for (j = 1; j &lt;= 5; j++) {&lt;br&gt;     if (j % 2 == 0) { Print &quot;*&quot; }&lt;br&gt;     else { Print &quot;-&quot; }&lt;br&gt;  }&lt;br&gt; System.out.println(); }</td>
<td>-<em>-</em>&lt;br&gt; -<em>-</em>&lt;br&gt; -<em>-</em></td>
<td>Prints asterisks in even columns, dashes in odd columns.</td>
</tr>
<tr>
<td>for (i = 1; i &lt;= 3; i++) { &lt;br&gt;  for (j = 1; j &lt;= 5; j++) {&lt;br&gt;     if ((i + j) % 2 == 0) { Print &quot;*&quot; }&lt;br&gt;     else { Print &quot; &quot; }&lt;br&gt;  }&lt;br&gt; System.out.println(); }</td>
<td>* * *&lt;br&gt; * *&lt;br&gt; * * *</td>
<td>Prints a checkerboard pattern.</td>
</tr>
</tbody>
</table>

### Self Check

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

10. What is the value of \( n \) after the following nested loops?

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

### Manipulating the Pixels in an Image

This Worked Example shows how to use nested loops for manipulating the pixels in an image. The outer loop traverses the rows of the image, and the inner loop accesses each pixel of a row.
A simulation program uses the computer to simulate an activity in the real world (or an imaginary one). Simulations are commonly used for predicting climate change, analyzing traffic, picking stocks, and many other applications in science and business. In many simulations, one or more loops are used to modify the state of a system and observe the changes.

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 5.) Buffon conjectured that the quotient \( \text{tries/hits} \) approximates \( \pi \).

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:

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>nextInt(n)</td>
<td>A random integer between the integers 0 (inclusive) and n (exclusive)</td>
</tr>
<tr>
<td>nextDouble()</td>
<td>A random floating-point number between 0 (inclusive) and 1 (exclusive)</td>
</tr>
</tbody>
</table>

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

```java
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.

If you call `nextInt` ten times, you get a random sequence of numbers similar to the following:

6 5 6 3 2 6 3 4 4 1

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

\[
\begin{array}{cccccccc}
6 & 5 & 6 & 3 & 2 & 6 & 3 & 4 \\
\end{array}
\]
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 $y_{\text{low}}$ 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 $\alpha$ 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 $y_{\text{high}}$ is at least 2. See Figure 6.

For the program that carries out the simulation of the needle experiment, see ch06/random2/ in your source code, or view it in WileyPLUS.

The point of this program is not to compute $\pi$—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.

11. How do you use a random number generator to simulate the toss of a coin?
12. Why is the NeedleSimulator program not an efficient method for computing π?

Special Topic 6.5
Loop Invariants

Special Topic 6.5 shows how you can use the technique of loop invariants to prove that a loop will always compute the correct result.

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.

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 (eclipse.org). Other integrated environments, such as BlueJ, also include debuggers. A free standalone debugger called JSwat is available from www.bluemarsh.com/java/jswat.
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.

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 7). 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.

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

**Figure 7** Stopping at a Breakpoint
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 debugger then displays the contents of the variable. If all variables contain what you expected, you can run the program until the next point where you want to stop.

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 8).

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.

For example, suppose the current line is

```java
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:

```java
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.

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

A debugger can be an effective tool for finding and removing bugs in your program. However, it is no substitute for good design and careful programming. If the debugger does not find any errors, it does not mean that your program is bug-free. Testing and debugging can only show the presence of bugs, not their absence.

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

14. 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?

**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 it 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 numbers, 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.
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.

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 aside some time and think first. Have a calculator handy to make independent 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.

---

**A Sample Debugging Session**

This Worked Example shows how to find bugs in an algorithm for counting the syllables of a word.

[Available online in WileyPLUS and at www.wiley.com/college/horstmann.]
Chapter 6 Iteration

Random Fact 6.1

The First Bug

According to legend, the first bug was one found in 1947 in the Mark II, a huge electromechanical 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 figure), it appears as if the term “bug” had already been in active use at the time.

The First Bug

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

Summary of Learning Objectives

Explain the flow of execution in a loop.
- A while statement executes a block of code repeatedly. A condition controls for how long the loop is executed.
- An off-by-one error is a common error when programming loops. Think through simple test cases to avoid this type of error.

Use for loops to implement counting loops.
- You use a for loop when a variable runs from a starting to an ending value with a constant increment or decrement.
- Make a choice between symmetric and asymmetric loop bounds.
- Count the number of iterations to check that your for loop is correct.

Implement loops that process a data set until a sentinel value is encountered.
- 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.

Use nested loops to implement multiple levels of iterations.
- When the body of a loop contains another loop, the loops are nested. A typical use of nested loops is printing a table with rows and columns.
Apply loops to the implementation of simulations that involve random values.

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

Use a debugger to locate errors in a running program.

- A debugger is a program that you can use to execute another program and analyze its run-time behavior.
- You can make effective use of a debugger by mastering just three concepts: breakpoints, single-stepping, and inspecting variables.
- When a debugger executes a program, the execution is suspended when-ever a breakpoint is reached.
- The single-step command executes the program one line at a time.
- A debugger can be used only to analyze the presence of bugs, not to show that a program is bug-free.
- Use the divide-and-conquer technique to locate the point of failure of a program.
- During debugging, compare the actual contents of variables against the values you know they should have.

Classes, Objects, and Methods Introduced in this Chapter

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

Media Resources

- **Worked Example**  Credit Card Processing
- **Worked Example**  Manipulating the Pixels in an Image
- **Worked Example**  A Sample Debugging Session
- Lab Exercises
  - **Animation**  Tracing a Loop
  - **Animation**  The for Loop
  - Practice Quiz
  - Code Completion Exercises

Review Exercises

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

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

```java
for (int i = 0; i < 10; i++)
{
```