

CHAPTER 6

LOOPS



CHAPTER GOALS

To implement while, for, and do loops
To hand-trace the execution of a program
To learn to use common loop algorithms
To understand nested loops
To implement programs that read and process data sets
To use a computer for simulations
To learn about the debugger

CHAPTER CONTENTS

6.1 THE WHILE LOOP 242

Syntax 6.1: while Statement 243

Common Error 6.1: Don't Think "Are We There Yet?" 247

Common Error 6.2: Infinite Loops 248

Common Error 6.3: Off-by-One Errors 248

6.2 PROBLEM SOLVING: HAND-TRACING 249

Computing & Society 6.1: Software Piracy 253

6.3 THE FOR LOOP 254

Syntax 6.2: for Statement 254

Programming Tip 6.1: Use for Loops for Their Intended Purpose Only 259

Programming Tip 6.2: Choose Loop Bounds That Match Your Task 260

Programming Tip 6.3: Count Iterations 260

Special Topic 6.1: Variables Declared in a for Loop Header 261

6.4 THE DO LOOP 262

Programming Tip 6.4: Flowcharts for Loops 263

6.5 APPLICATION: PROCESSING SENTINEL VALUES 263

Special Topic 6.2: Redirection of Input and Output 266

Special Topic 6.3: The "Loop and a Half" Problem 266

Special Topic 6.4: The break and continue Statements 267


6.6 PROBLEM SOLVING: STORYBOARDS 269

6.7 COMMON LOOP ALGORITHMS 272

How To 6.1: Writing a Loop 276

Worked Example 6.1: Credit Card Processing 


6.8 NESTED LOOPS 279

Worked Example 6.2: Manipulating the Pixels in an Image 

6.9 APPLICATION: RANDOM NUMBERS AND SIMULATIONS 283

6.10 USING A DEBUGGER 286

How To 6.2: Debugging 289

Worked Example 6.3: A Sample Debugging Session 

Computing & Society 6.2: The First Bug 291



In a loop, a part of a program is repeated over and over, until a specific goal is reached. Loops are important for calculations that require repeated steps and for processing input consisting of many data items. In this chapter, you will learn about loop statements in Java, as well as techniques for writing programs that process input and simulate activities in the real world.

6.1 The while Loop

In this section, you will learn about *loop statements* that repeatedly execute instructions until a goal has been reached.

Recall the investment problem from Chapter 1. You put \$10,000 into a bank account that earns 5 percent interest per year. How many years does it take for the account balance to be double the original investment?

In Chapter 1 we developed the following algorithm for this problem:

Start with a year value of 0, a column for the interest, and a balance of \$10,000.

year	interest	balance
0		\$10,000

Repeat the following steps while the balance is less than \$20,000.

Add 1 to the year value.

Compute the interest as $\text{balance} \times 0.05$ (i.e., 5 percent interest).

Add the interest to the balance.

Report the final year value as the answer.

You now know how to declare and update the variables in Java. What you don't yet know is how to carry out "Repeat steps while the balance is less than \$20,000".



Because the interest earned also earns interest, a bank balance grows exponentially.

In a particle accelerator, subatomic particles traverse a loop-shaped tunnel multiple times, gaining the speed required for physical experiments. Similarly, in computer science, statements in a loop are executed while a condition is true.



Figure 1 Flowchart of a while Loop

A loop executes instructions repeatedly while a condition is true.

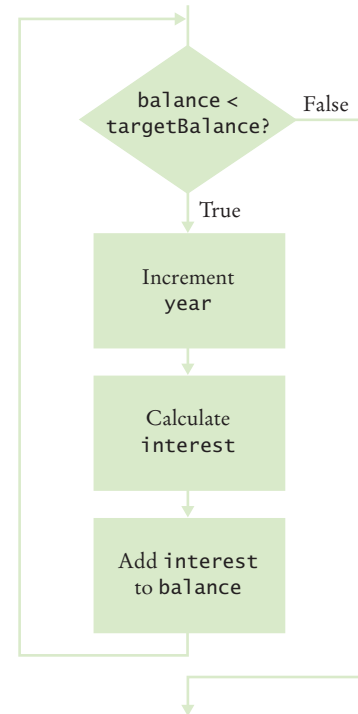
In Java, the while statement implements such a repetition (see Syntax 6.1). It has the form

```
while (condition)
{
    statements
}
```

As long as the condition remains true, the statements inside the while statement are executed. These statements are called the **body** of the while statement.

In our case, we want to increment the year counter and add interest while the balance is less than the target balance of \$20,000:

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




A while statement is an example of a **loop**. If you draw a flowchart, the flow of execution loops again to the point where the condition is tested (see Figure 1).

Syntax 6.1 while Statement

Syntax `while (condition)`
`{`
 statements
`}`

This variable is declared outside the loop and updated in the loop.

If the condition never becomes false, an infinite loop occurs.  See page 248.

This variable is created in each loop iteration.

```
double balance = 0;
```

```
.
```

```
.
```


```
.
```


```
{
```

```
    double interest = balance * RATE / 100;
```


```
    balance = balance + interest;
```


```
}
```

Beware of "off-by-one" errors in the loop condition.  See page 248.

Don't put a semicolon here!  See page 184.

These statements are executed while the condition is true.

Lining up braces is a good idea.  See page 184.

Braces are not required if the body contains a single statement, but it's good to always use them.  See page 184.

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:

```
while (balance < targetBalance)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
// interest no longer declared here
```

A new interest variable
is created in each iteration.

1 Check the loop condition

balance = 10000

year = 0

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

The condition is true

2 Execute the statements in the loop

balance = 10500

year = 1

interest = 500

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

3 Check the loop condition again

balance = 10500

year = 1

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

The condition is still true

⋮

4 After 15 iterations

balance = 20789.28

year = 15

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

The condition is
no longer true

5 Execute the statement following the loop

balance = 20789.28

year = 15

```
while (balance < targetBalance)
{
    year++;
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
System.out.println(year);
```

Figure 2
Execution of the
Investment Loop

In contrast, the `balance` and `year` variables were declared outside the loop body. That way, the same variable is used for all iterations of the loop.

Here is the program that solves the investment problem. Figure 2 illustrates the program's execution.

section_1/Investment.java

```

1  /**
2   * A class to monitor the growth of an investment that
3   * accumulates interest at a fixed annual rate.
4   */
5  public class Investment
6  {
7      private double balance;
8      private double rate;
9      private int year;
10
11     /**
12      * Constructs an Investment object from a starting balance and
13      * interest rate.
14      * @param aBalance the starting balance
15      * @param aRate the interest rate in percent
16      */
17     public Investment(double aBalance, double aRate)
18     {
19         balance = aBalance;
20         rate = aRate;
21         year = 0;
22     }
23
24     /**
25      * Keeps accumulating interest until a target balance has
26      * been reached.
27      * @param targetBalance the desired balance
28      */
29     public void waitForBalance(double targetBalance)
30     {
31         while (balance < targetBalance)
32         {
33             year++;
34             double interest = balance * rate / 100;
35             balance = balance + interest;
36         }
37     }
38
39     /**
40      * Gets the current investment balance.
41      * @return the current balance
42      */
43     public double getBalance()
44     {
45         return balance;
46     }
47
48     /**
49      * Gets the number of years this investment has accumulated
50      * interest.
51      * @return the number of years since the start of the investment

```

```

52     */
53     public int getYears()
54     {
55         return year;
56     }
57 }

```

section_1/InvestmentRunner.java

```

1  /**
2   * This program computes how long it takes for an investment
3   * to double.
4   */
5  public class InvestmentRunner
6  {
7      public static void main(String[] args)
8      {
9          final double INITIAL_BALANCE = 10000;
10         final double RATE = 5;
11         Investment invest = new Investment(INITIAL_BALANCE, RATE);
12         invest.waitForBalance(2 * INITIAL_BALANCE);
13         int years = invest.getYears();
14         System.out.println("The investment doubled after "
15                             + years + " years");
16     }
17 }

```

Program Run

The investment doubled after 15 years.



1. How many years does it take for the investment to triple? Modify the program and run it.
2. If the interest rate is 10 percent per year, how many years does it take for the investment to double? Modify the program and run it.
3. Modify the program so that the balance after each year is printed. How did you do that?
4. Suppose we change the program so that the condition of the while loop is `while (balance <= targetBalance)`. What is the effect on the program? Why?
5. What does the following loop print?

```

int n = 1;
while (n < 100)
{
    n = 2 * n;
    System.out.print(n + " ");
}

```

Practice It Now you can try these exercises at the end of the chapter: R6.1, R6.5, E6.13.

Table 1 while Loop Examples

Loop	Output	Explanation
<pre>i = 0; sum = 0; while (sum < 10) { i++; sum = sum + i; Print i and sum; }</pre>	<pre>1 1 2 3 3 6 4 10</pre>	When sum is 10, the loop condition is false, and the loop ends.
<pre>i = 0; sum = 0; while (sum < 10) { i++; sum = sum - i; Print i and sum; }</pre>	<pre>1 -1 2 -3 3 -6 4 -10 . . .</pre>	Because sum never reaches 10, this is an “infinite loop” (see Common Error 6.2 on page 248).
<pre>i = 0; sum = 0; while (sum < 0) { i++; sum = sum - i; Print i and sum; }</pre>	(No output)	The statement <code>sum < 0</code> is false when the condition is first checked, and the loop is never executed.
<pre>i = 0; sum = 0; while (sum >= 10) { i++; sum = sum + i; Print i and sum; }</pre>	(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 (see Common Error 6.1 on page 247).
<pre>i = 0; sum = 0; while (sum < 10) ; { i++; sum = sum + i; Print i and sum; }</pre>	(No output, program does not terminate)	Note the semicolon before the <code>{</code> . This loop has an empty body. It runs forever, checking whether <code>sum < 10</code> and doing nothing in the body.

Common Error 6.1



Don't Think “Are We There Yet?”

When doing something repetitive, most of us want to know when we are done. For example, you may think, “I want to get at least \$20,000,” and set the loop condition to

```
balance >= targetBalance
```

But the while loop thinks the opposite: How long am I allowed to keep going? The correct loop condition is

```
while (balance < targetBalance)
```

In other words: “Keep at it while the balance is less than the target.”

*When writing a loop condition, don't ask, “Are we there yet?”
The condition determines how long the loop will keep going.*



Common Error 6.2

**Infinite Loops**

A very 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 program, 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 + C. On others, you can close the window in which the program runs.

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

```
int year = 1;
while (year <= 20)
{
    double interest = balance * RATE / 100;
    balance = balance + interest;
}
```

Here the programmer forgot to add a `year++` command in the loop. As a result, the `year` always stays at 1, 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 year = 20;
while (year > 0)
{
    double interest = balance * RATE / 100;
    balance = balance + interest;
    year++;
}
```

The `year` 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, `year` is always larger than 0, and the loop never ends. (Actually, `year` may eventually exceed the largest representable positive integer and *wrap around* to a negative number. Then the loop ends—of course, with a completely wrong result.)



Like this hamster who can't stop running in the treadmill, an infinite loop never ends.

Common Error 6.3

**Off-by-One Errors**

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

```
int year = 0;
while (balance < targetBalance)
{
    year++;
    balance = balance * (1 + RATE / 100);
}
System.out.println("The investment doubled after "
    + year + " years.");
```

Should `year` start at 0 or at 1? Should you test for `balance < targetBalance` or for `balance <= targetBalance`? 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—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 your decisions.

Should year 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 year each time. Hence year must start at 0, not at 1.

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

year	balance
0	\$100
1	\$150
2	\$225

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? This is harder to figure out, because it is rare for the balance to be exactly twice the initial balance. There is one case when this happens, namely when the interest is 100 percent. The loop executes once. Now year is 1, and balance is exactly equal to `2 * INITIAL_BALANCE`. Has the investment doubled after one year? It has. Therefore, the loop should not execute again. If the test condition is `balance < targetBalance`, the loop stops, as it should. If the test condition had been `balance <= targetBalance`, the loop would have executed once more.

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

6.2 Problem Solving: Hand-Tracing

Hand-tracing is a simulation of code execution in which you step through instructions and track the values of the variables.

In Programming Tip 5.5, you learned about the method of hand-tracing. When you hand-trace code or pseudocode, you write the names of the variables on a sheet of paper, mentally execute each step of the code, and update the variables.

It is best to have the code written or printed on a sheet of paper. Use a marker, such as a paper clip, to mark the current line. Whenever a variable changes, cross out the old value and write the new value below. When a program produces output, also write down the output in another column.

Consider this example. What value is displayed?

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

There are three variables: `n`, `sum`, and `digit`.

<code>n</code>	<code>sum</code>	<code>digit</code>

The first two variables are initialized with 1729 and 0 before the loop is entered.

```

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

```

<code>n</code>	<code>sum</code>	<code>digit</code>
1729	0	

Because `n` is greater than zero, enter the loop. The variable `digit` is set to 9 (the remainder of dividing 1729 by 10). The variable `sum` is set to $0 + 9 = 9$.

```

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

```

<code>n</code>	<code>sum</code>	<code>digit</code>
1729	0	
	9	9

Finally in this iteration, `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.

```

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

```

<code>n</code>	<code>sum</code>	<code>digit</code>
1729	0	
172	9	9

Now check the loop condition again.

```

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

```

Because n is still greater than zero, repeat the loop. Now $digit$ becomes 2, sum is set to $9 + 2 = 11$, and n is set to 17.

n	sum	digit
1729	0	
172	9	9
17	11	2

Repeat the loop once again, setting $digit$ to 7, sum to $11 + 7 = 18$, and n to 1.

n	sum	digit
1729	0	
172	9	9
17	11	2
1	18	7

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

n	sum	digit
1729	0	
172	9	9
17	11	2
1	18	7
0	19	1

```

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

```

Because n equals zero, this condition is not true.

The condition $n > 0$ is now false. Continue with the statement after the loop.

```

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

```

n	sum	digit	output
1729	0		
172	9	9	
17	11	2	
1	18	7	
0	19	1	19



This statement is an output statement. The value that is output is the value of sum , which is 19.

Of course, you can get the same answer by just running the code. However, hand-tracing can give you an *insight* that you would not get if you simply ran the code. 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 .

Hand-tracing can help you understand how an unfamiliar algorithm works.

Hand-tracing can show errors in code or pseudocode.

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 numbers.)

Hand-tracing does not just help you understand code that works correctly. It is a powerful technique for finding errors in your code. When a program behaves in a way that you don't expect, get out a sheet of paper and track the values of the variables as you mentally step through the code.

You don't need a working program to do hand-tracing. You can hand-trace pseudocode. In fact, it is an excellent idea to hand-trace your pseudocode before you go to the trouble of translating it into actual code, to confirm that it works correctly.



6. Hand-trace the following code, showing the value of n and the output.

```
int n = 5;
while (n >= 0)
{
    n--;
    System.out.print(n);
}
```

7. Hand-trace the following code, showing the value of n and the output. What potential error do you notice?

```
int n = 1;
while (n <= 3)
{
    System.out.print(n + ", ");
    n++;
}
```

8. Hand-trace the following code, assuming that a is 2 and n is 4. Then explain what the code does for arbitrary values of a and n .

```
int r = 1;
int i = 1;
while (i <= n)
{
    r = r * a;
    i++;
}
```

9. Trace the following code. What error do you observe?

```
int n = 1;
while (n != 50)
{
    System.out.println(n);
    n = n + 10;
}
```

10. The following pseudocode is intended to count the number of digits in the number n :

```
count = 1
temp = n
while (temp > 10)
    Increment count.
    Divide temp by 10.0.
```

Trace the pseudocode for $n = 123$ and $n = 100$. What error do you find?

Practice It Now you can try these exercises at the end of the chapter: R6.3, R6.6.



Computing & Society 6.1 Software Piracy

As you read this, you will have written a few computer programs and experienced firsthand how much effort it takes to write even the humblest of programs. Writing a real software product, such as a financial application or a computer game, takes a lot of time and money. Few people, and fewer companies, are going to spend that kind of time and money if they don't have a reasonable chance to make more money from their effort. (Actually, some companies give away their software in the hope that users will upgrade to more elaborate paid versions. Other companies give away the software that enables users to read and use files but sell the software needed to create those files. Finally, there are individuals who donate their time, out of enthusiasm, and produce programs that you can copy freely.)

When selling software, a company must rely on the honesty of its customers. It is an easy matter for an unscrupulous person to make copies of computer programs without paying for them. In most countries that is illegal. Most governments provide legal protection, such as copyright laws and patents, to encourage the development of new products. Countries that tolerate widespread piracy have found that they have an ample cheap supply of foreign software, but no local manufacturers willing to design good soft-

ware for their own citizens, such as word processors in the local script or financial programs adapted to the local tax laws.

When a mass market for software first appeared, vendors were enraged by the money they lost through piracy. They tried to fight back by various schemes to ensure that only the legitimate owner could use the software, such as *dongles*—devices that must be attached to a printer port before the software will run. Legitimate users hated these measures. They paid for the software, but they had to suffer through inconveniences, such as having multiple dongles stick out from their computer. In the United States, market pressures forced most vendors to give up on these copy protection schemes, but they are still commonplace in other parts of the world.

Because it is so easy and inexpensive to pirate software, and the chance of being found out is minimal, you have to make a moral choice for yourself. If a package that you would really like to have is too expensive for your budget, do you steal it, or do you stay honest and get by with a more affordable product?

Of course, piracy is not limited to software. The same issues arise for other digital products as well. You may have had the opportunity to obtain copies of songs or movies without

payment.

Or you may have been frustrated by a copy protection device on your music player that made it difficult for you to listen to songs that you paid for. Admittedly, it can

be difficult to have a lot of sympathy for a musical ensemble whose publisher charges a lot of money for what seems to have been very little effort on their part, at least when compared to the effort that goes into designing and implementing a software package. Nevertheless, it seems only fair that artists and authors receive some compensation for their efforts. How to pay artists, authors, and programmers fairly, without burdening honest customers, is an unsolved problem at the time of this writing, and many computer scientists are engaged in research in this area.



6.3 The for Loop

The for loop is used when a value runs from a starting point to an ending point with a constant increment or decrement.

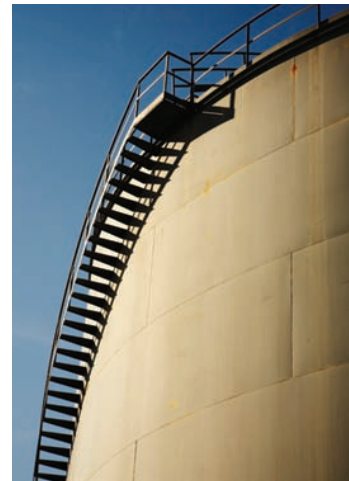
It often happens that you want to execute a sequence of statements a given number of times. You can use a `while` loop that is controlled by a counter, as in the following example:

```
int counter = 1; // Initialize the counter
while (counter <= 10) // Check the counter
{
    System.out.println(counter);
    counter++; // Update the counter
}
```

Because this loop type is so common, there is a special form for it, called the `for` loop (see Syntax 6.2).

```
for (int counter = 1; counter <= 10; counter++)
{
    System.out.println(counter);
}
```

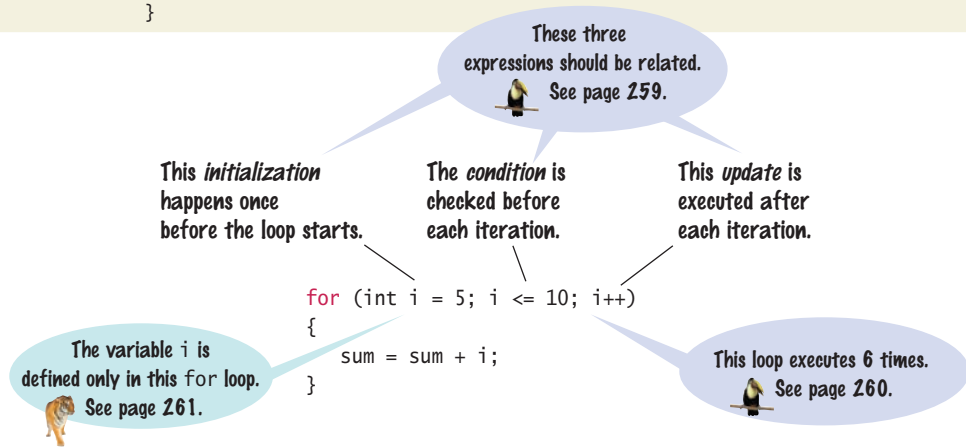
Some people call this loop *count-controlled*. In contrast, the `while` loop of the preceding section can be called an *event-controlled* loop because it executes until an event occurs; namely that the balance reaches the target. Another commonly used term for a count-controlled loop is *definite*. You know from the outset that the loop body will be executed a definite number of times; ten times in our example. In contrast, you do not know how many iterations it takes to accumulate a target balance. Such a loop is called *indefinite*.



You can visualize the `for` loop as an orderly sequence of steps.

Syntax 6.2 for Statement

```
Syntax  for (initialization; condition; update)
{
    statements
}
```





The for loop neatly groups the initialization, condition, and update expressions together. However, it is important to realize that these expressions are not executed together (see Figure 3).

- The initialization is executed once, before the loop is entered. **1**
- The condition is checked before each iteration. **2 5**
- The update is executed after each iteration. **4**

1 Initialize counter counter = 1	<pre>for (int counter = 1; counter <= 10; counter++) { System.out.println(counter); }</pre>
2 Check condition counter = 1	<pre>for (int counter = 1; counter <= 10; counter++) { System.out.println(counter); }</pre>
3 Execute loop body counter = 1	<pre>for (int counter = 1; counter <= 10; counter++) { System.out.println(counter); }</pre>
4 Update counter counter = 2	<pre>for (int counter = 1; counter <= 10; counter++) { System.out.println(counter); }</pre>
5 Check condition again counter = 2	<pre>for (int counter = 1; counter <= 10; counter++) { System.out.println(counter); }</pre>

Figure 3
Execution of a
for Loop

A for loop can count down instead of up:

```
for (int counter = 10; counter >= 0; counter--) . . .
```

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

```
for (int counter = 0; counter <= 10; counter = counter + 2) . . .
```

See Table 2 on page 258 for additional variations.

So far, we have always declared the counter variable in the loop initialization:

```
for (int counter = 1; counter <= 10; counter++)
{
    . . .
}
// counter no longer declared here
```


Such a variable is declared for all iterations of the loop, but you cannot use it after the loop. If you declare the counter variable before the loop, you can continue to use it after the loop:

```
int counter;
for (counter = 1; counter <= 10; counter++)
{
    . . .
}
// counter still declared here
```

A common use of the for loop is to traverse all characters of a string:

```
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.

Here is another typical use of the for loop. We want to compute the growth of our savings account over a period of years, as shown in this table:

Year	Balance
1	10500.00
2	11025.00
3	11576.25
4	12155.06
5	12762.82

The for loop pattern applies because the variable *year* starts at 1 and then moves in constant increments until it reaches the target:

```
for (int year = 1; year <= numberOfYears; year++)
{
    Update balance.
}
```

Following is the complete program. Figure 4 shows the corresponding flowchart.

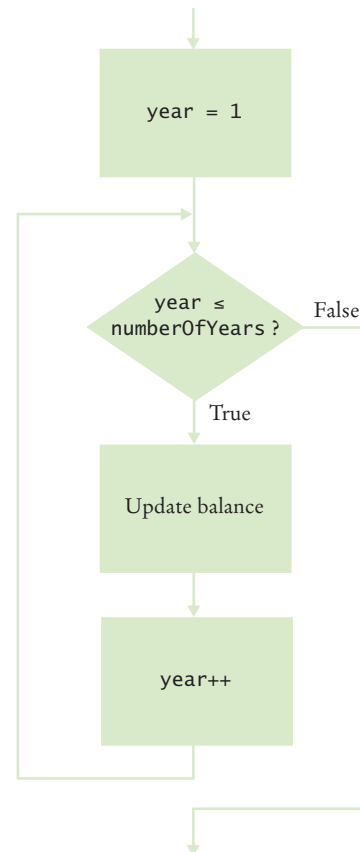


Figure 4 Flowchart of a for Loop

section_3/Investment.java

```

1  /**
2   * A class to monitor the growth of an investment that
3   * accumulates interest at a fixed annual rate.
4   */
5  public class Investment
6  {
7      private double balance;
8      private double rate;
9      private int year;
10
11     /**
12      * Constructs an Investment object from a starting balance and
13      * interest rate.
14      * @param aBalance the starting balance
15      * @param aRate the interest rate in percent
16      */
17     public Investment(double aBalance, double aRate)
18     {
19         balance = aBalance;
20         rate = aRate;
21         year = 0;
22     }
23
24     /**
25      * Keeps accumulating interest until a target balance has
26      * been reached.
27      * @param targetBalance the desired balance
28      */
29     public void waitForBalance(double targetBalance)
30     {
31         while (balance < targetBalance)
32         {
33             year++;
34             double interest = balance * rate / 100;
35             balance = balance + interest;
36         }
37     }
38
39     /**
40      * Keeps accumulating interest for a given number of years.
41      * @param numberOfYears the number of years to wait
42      */
43     public void waitYears(int numberOfYears)
44     {
45         for (int i = 1; i <= numberOfYears; i++)
46         {
47             double interest = balance * rate / 100;
48             balance = balance + interest;
49         }
50         year = year + n;
51     }
52
53     /**
54      * Gets the current investment balance.
55      * @return the current balance
56      */
57     public double getBalance()
58     {

```

```

59     return balance;
60 }
61
62 /**
63  * Gets the number of years this investment has accumulated
64  * interest.
65  * @return the number of years since the start of the investment
66  */
67 public int getYears()
68 {
69     return year;
70 }
71 }

```

section_3/InvestmentRunner.java

```

1  /**
2   * This program computes how much an investment grows in
3   * a given number of years.
4   */
5  public class InvestmentRunner
6  {
7      public static void main(String[] args)
8      {
9          final double INITIAL_BALANCE = 10000;
10         final double RATE = 5;
11         final int YEARS = 20;
12         Investment invest = new Investment(INITIAL_BALANCE, RATE);
13         invest.waitYears(YEARS);
14         double balance = invest.getBalance();
15         System.out.printf("The balance after %d years is %.2f\n",
16             YEARS, balance);
17     }
18 }

```

Program Run

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

Table 2 for Loop Examples

Loop	Values of i	Comment
for (i = 0; i <= 5; i++)	0 1 2 3 4 5	Note that the loop is executed 6 times. (See Programming Tip 6.3 on page 260.)
for (i = 5; i >= 0; i--)	5 4 3 2 1 0	Use i-- for decreasing values.
for (i = 0; i < 9; i = i + 2)	0 2 4 6 8	Use i = i + 2 for a step size of 2.
for (i = 0; i != 9; i = i + 2)	0 2 4 6 8 10 12 14 ... (infinite loop)	You can use < or <= instead of != to avoid this problem.
for (i = 1; i <= 20; i = i * 2)	1 2 4 8 16	You can specify any rule for modifying i, such as doubling it in every step.
for (i = 0; i < str.length(); i++)	0 1 2 ... until the last valid index of the string str	In the loop body, use the expression str.charAt(i) to get the ith character.

SELF CHECK



11. Write the for loop of the `Investment` class as a `while` loop.
12. How many numbers does this loop print?

```
for (int n = 10; n >= 0; n--)
{
    System.out.println(n);
}
```
13. Write a for loop that prints all even numbers between 10 and 20 (inclusive).
14. Write a for loop that computes the sum of the integers from 1 to `n`.
15. How would you modify the `InvestmentRunner.java` program to print the balances after 20, 40, ..., 100 years?

Practice It Now you can try these exercises at the end of the chapter: R6.4, R6.10, E6.8, E6.12.

Programming Tip 6.1

**Use for Loops for Their Intended Purpose Only**

A for loop is an *idiom* for a loop of a particular form. A value runs from the start to the end, with a constant increment or decrement.

The compiler won't check whether the initialization, condition, and update expressions are related. For example, the following loop is legal:

```
// Confusing—unrelated expressions
for (System.out.print("Inputs: "); in.hasNextDouble(); sum = sum + x)
{
    x = in.nextDouble();
}
```

However, programmers reading such a for loop will be confused because it does not match their expectations. Use a `while` loop for iterations that do not follow the for idiom.

You should also be careful not to update the loop counter in the body of a for loop. Consider the following example:

```
for (int counter = 1; counter <= 100; counter++)
{
    if (counter % 10 == 0) // Skip values that are divisible by 10
    {
        counter++; // Bad style—you should not update the counter in a for loop
    }
    System.out.println(counter);
}
```

Updating the counter inside a for loop is confusing because the counter is updated *again* at the end of the loop iteration. In some loop iterations, `counter` is incremented once, in others twice. This goes against the intuition of a programmer who sees a for loop.

If you find yourself in this situation, you can either change from a for loop to a `while` loop, or implement the “skipping” behavior in another way. For example:

```
for (int counter = 1; counter <= 100; counter++)
{
    if (counter % 10 != 0) // Skip values that are divisible by 10
    {
        System.out.println(counter);
    }
}
```

Programming Tip 6.2

**Choose Loop Bounds That Match Your Task**

Suppose you want to print line numbers that go from 1 to 10. Of course, you will use a loop:

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

The values for i are bounded by the relation $1 \leq i \leq 10$. Because there are \leq on both bounds, the bounds are called **symmetric bounds**.

When traversing the characters in a string, it is more natural to use the bounds

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

In this loop, i traverses all valid positions in the string. You can access the i th character as `str.charAt(i)`. The values for i are bounded by $0 \leq i < \text{str.length}()$, with a \leq to the left and a $<$ to the right. That is appropriate, because `str.length()` is not a valid position. Such bounds are called **asymmetric bounds**.

In this case, it is not a good idea to use symmetric bounds:

```
for (int i = 0; i <= str.length() - 1; i++) // Use < instead
```

The asymmetric form is easier to understand.

Programming Tip 6.3

**Count Iterations**

Finding the correct lower and upper bounds for an iteration can be confusing. Should you start at 0 or at 1? Should you use `<= 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

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

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

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

is executed $b - a + 1$ times. That “+1” is the source of many programming errors.

For example,

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

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 by looking at a fence. Each section has one fence post to the left, and there is a final post on the right of the last section. Forgetting to count the last value is often called a “fence post error”.



How many posts do you need for a fence with four sections? It is easy to be “off by one” with problems such as this one.

Special Topic 6.1

**Variables Declared 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 declare 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 Special Topic 6.4 on page 267). 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 declare 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 Programming Tip 6.2 on page 260). 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--;
}
```

6.4 The do Loop

The do loop is appropriate when the loop body must be executed at least once.

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

```
do
{
    statements
}
while (condition);
```

The body of the do loop is executed first, then the condition is tested.

Some people call such a loop a *post-test loop* because the condition is tested after completing the loop body. In contrast, *while* and *for* loops are *pre-test loops*. In those loop types, the condition is tested before entering the loop body.

A typical example for a do loop is input validation. Suppose you ask a user to enter a value < 100. If the user doesn't pay attention and enters a larger value, you ask again, until the value is correct. Of course, you cannot test the value until the user has entered it. This is a perfect fit for the do loop (see Figure 5):

```
int value;
do
{
    System.out.print("Enter an integer < 100: ");
    value = in.nextInt();
}
while (value >= 100);
```

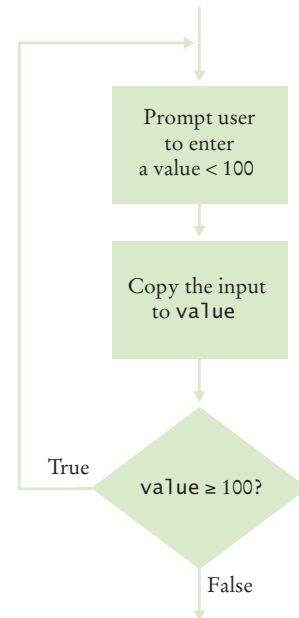


Figure 5 Flowchart of a do Loop

SELF CHECK



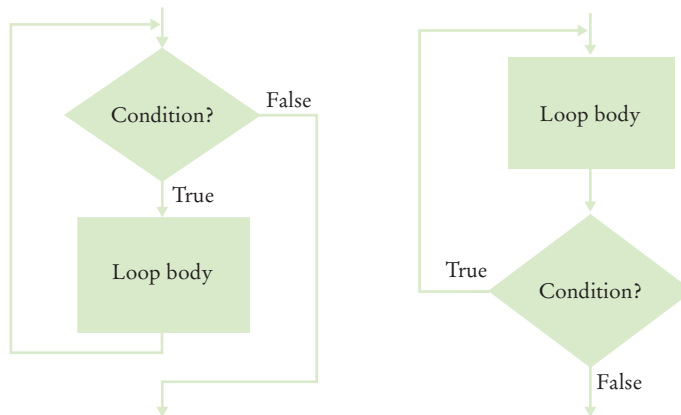
16. Suppose that we want to check for inputs that are at least 0 and at most 100. Modify the do loop for this check.
17. Rewrite the input check do loop using a *while* loop. What is the disadvantage of your solution?
18. Suppose Java didn't have a do loop. Could you rewrite any do loop as a *while* loop?
19. Write a do loop that reads integers and computes their sum. Stop when reading the value 0.
20. Write a do loop that reads integers and computes their sum. Stop when reading a zero or the same value twice in a row. For example, if the input is 1 2 3 4 4, then the sum is 14 and the loop stops.

Practice It Now you can try these exercises at the end of the chapter: R6.9, R6.16, R6.17.

Programming Tip 6.4

**Flowcharts for Loops**

In Section 5.5, you learned how to use flowcharts to visualize the flow of control in a program. There are two types of loops that you can include in a flowchart; they correspond to a `while` loop and a `do` loop in Java. They differ in the placement of the condition—either before or after the loop body.



As described in Section 5.5, you want to avoid “spaghetti code” in your flowcharts. For loops, that means that you never want to have an arrow that points inside a loop body.

6.5 Application: Processing Sentinel Values

A sentinel value denotes the end of a data set, but it is not part of the data.

In this section, you will learn how to write loops that read and process a sequence of input values.

Whenever you read a sequence of inputs, you need to have some method of indicating the end of the sequence. Sometimes you are lucky and no input value can be zero. Then you can prompt the user to keep entering numbers, or 0 to finish the sequence. If zero is allowed but negative numbers are not, you can use `-1` to indicate termination.

Such a value, which is not an actual input, but serves as a signal for termination, is called a **sentinel**.

Let’s put this technique to work in a program that computes the average of a set of salary values. In our sample program, we will use `-1` as a sentinel. An employee would surely not work for a negative salary, but there may be volunteers who work for free.



In the military, a sentinel guards a border or passage. In computer science, a sentinel value denotes the end of an input sequence or the border between input sequences.

Inside the loop, we read an input. If the input is not `-1`, we process it. In order to compute the average, we need the total sum of all salaries, and the number of inputs.

```
salary = in.nextDouble();
if (salary != -1)
{
    sum = sum + salary;
    count++;
}
```

We stay in the loop while the sentinel value is not detected.

```
while (salary != -1)
{
    . . .
}
```

There is just one problem: When the loop is entered for the first time, no data value has been read. We must make sure to initialize `salary` with some value other than the sentinel:

```
double salary = 0;
// Any value other than -1 will do
```

After the loop has finished, we compute and print the average. Here is the complete program:

section_5/SentinelDemo.java

```
1  import java.util.Scanner;
2
3  /**
4   * This program prints the average of salary values that are terminated with a sentinel.
5   */
6  public class SentinelDemo
7  {
8      public static void main(String[] args)
9      {
10         double sum = 0;
11         int count = 0;
12         double salary = 0;
13         System.out.print("Enter salaries, -1 to finish: ");
14         Scanner in = new Scanner(System.in);
15
16         // Process data until the sentinel is entered
17
18         while (salary != -1)
19         {
20             salary = in.nextDouble();
21             if (salary != -1)
22             {
23                 sum = sum + salary;
24                 count++;
25             }
26         }
27
28         // Compute and print the average
29
30         if (count > 0)
31         {
32             double average = sum / count;
```

```

33         System.out.println("Average salary: " + average);
34     }
35     else
36     {
37         System.out.println("No data");
38     }
39 }
40 }

```

Program Run

```

Enter salaries, -1 to finish: 10 10 40 -1
Average salary: 20

```

You can use a Boolean variable to control a loop. Set the variable before entering the loop, then set it to the opposite to leave the loop.

Some programmers don't like the "trick" of initializing the input variable with a value other than the sentinel. Another approach is to use a Boolean variable:

```

System.out.print("Enter salaries, -1 to finish: ");
boolean done = false;
while (!done)
{
    value = in.nextDouble();
    if (value == -1)
    {
        done = true;
    }
    else
    {
        Process value.
    }
}

```

Special Topic 6.4 on page 267 shows an alternative mechanism for leaving such a loop.

Now consider the case in which any number (positive, negative, or zero) can be an acceptable input. In such a situation, you must use a sentinel that is not a number (such as the letter Q). As you have seen in Section 5.8, the condition

```
in.hasNextDouble()
```

is false if the next input is not a floating-point number. Therefore, you can read and process a set of inputs with the following loop:

```

System.out.print("Enter values, Q to quit: ");
while (in.hasNextDouble())
{
    value = in.nextDouble();
    Process value.
}

```

SELF CHECK



21. What does the `SentinelDemo.java` program print when the user immediately types `-1` when prompted for a value?
22. Why does the `SentinelDemo.java` program have *two* checks of the form `salary != -1`?
23. What would happen if the declaration of the `salary` variable in `SentinelDemo.java` was changed to `double salary = -1;`?

24. In the last example of this section, we prompt the user “Enter values, Q to quit: ” What happens when the user enters a different letter?

25. What is wrong with the following loop for reading a sequence of values?

```
System.out.print("Enter values, Q to quit: ");
do
{
    double value = in.nextDouble();
    sum = sum + value;
    count++;
}
while (in.hasNextDouble());
```

Practice It Now you can try these exercises at the end of the chapter: R6.13, E6.17, E6.18.

Special Topic 6.2



Redirection of Input and Output

Consider the `SentinelDemo` program that computes the average value of an input sequence. If you use such a program, then it is quite likely that you already have the values in a file, and it seems a shame that you have to type them all in again. The command line interface of your operating system provides a way to link a file to the input of a program, as if all the characters in the file had actually been typed by a user. If you type

```
java SentinelDemo < numbers.txt
```

the program is executed, but it no longer expects input from the keyboard. All input commands get their input from the file `numbers.txt`. This process is called **input redirection**.

Input redirection is an excellent tool for testing programs. When you develop a program and fix its bugs, it is boring to keep entering the same input every time you run the program. Spend a few minutes putting the inputs into a file, and use redirection.

You can also redirect output. In this program, that is not terribly useful. If you run

```
java SentinelDemo < numbers.txt > output.txt
```

the file `output.txt` contains the input prompts and the output, such as

```
Enter salaries, -1 to finish: Enter salaries, -1 to finish:
Enter salaries, -1 to finish: Enter salaries, -1 to finish:
Average salary: 15
```

However, redirecting output is obviously useful for programs that produce lots of output. You can format or print the file containing the output.

Use input redirection to read input from a file. Use output redirection to capture program output in a file.

Special 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.equals("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()).equals("Q"))
{
    Process data.
}

```

The expression

```
(input = in.next()).equals("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 Special Topic 6.4 on page 267).

```

public void processInput(Scanner in)
{
    while (true)
    {
        String input = in.next();
        if (input.equals("Q"))
        {
            return;
        }
        Process data.
    }
}

```

Special Topic 6.4



The break and continue Statements

You already encountered the `break` statement in Special 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.equals("Q"))
{
    break;
}
double x = Double.parseDouble(input);
data.add(x);
}

```

A loop with break statements can be difficult to understand because you have to look closely to find out how to exit the loop. 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 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.equals("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.

6.6 Problem Solving: Storyboards

A storyboard consists of annotated sketches for each step in an action sequence.

Developing a storyboard helps you understand the inputs and outputs that are required for a program.

When you design a program that interacts with a user, you need to make a plan for that interaction. What information does the user provide, and in which order? What information will your program display, and in which format? What should happen when there is an error? When does the program quit?

This planning is similar to the development of a movie or a computer game, where *storyboards* are used to plan action sequences. A storyboard is made up of panels that show a sketch of each step. Annotations explain what is happening and note any special situations. Storyboards are also used to develop software—see Figure 6.

Making a storyboard is very helpful when you begin designing a program. You need to ask yourself which information you need in order to compute the answers that the program user wants. You need to decide how to present those answers. These are important considerations that you want to settle before you design an algorithm for computing the answers.

Let's look at a simple example. We want to write a program that helps users with questions such as “How many tablespoons are in a pint?” or “How many inches are 30 centimeters?”

What information does the user provide?

- The quantity and unit to convert from
- The unit to convert to

What if there is more than one quantity? A user may have a whole table of centimeter values that should be converted into inches.

What if the user enters units that our program doesn't know how to handle, such as ångström?

What if the user asks for impossible conversions, such as inches to gallons?

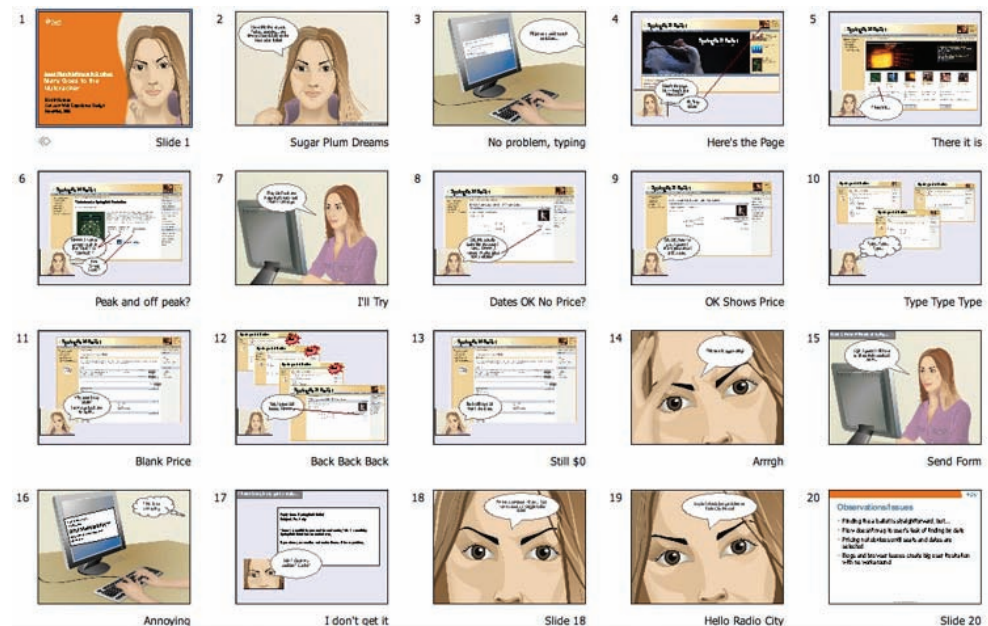


Figure 6
Storyboard for the
Design of a Web
Application

Let's get started with a storyboard panel. It is a good idea to write the user inputs in a different color. (Underline them if you don't have a color pen handy.)

Converting a Sequence of Values

```

What unit do you want to convert from? cm
What unit do you want to convert to? in
Enter values, terminated by zero ————— Allows conversion of multiple values
30
30 cm = 11.81 in ————— Format makes clear what got converted
100
100 cm = 39.37 in
0
What unit do you want to convert from?
  
```

The storyboard shows how we deal with a potential confusion. A user who wants to know how many inches are 30 centimeters may not read the first prompt carefully and specify inches. But then the output is “30 in = 76.2 cm”, alerting the user to the problem.

The storyboard also raises an issue. How is the user supposed to know that “cm” and “in” are valid units? Would “centimeter” and “inches” also work? What happens when the user enters a wrong unit? Let's make another storyboard to demonstrate error handling.

Handling Unknown Units (needs improvement)

```

What unit do you want to convert from? cm
What unit do you want to convert to? inches
Sorry, unknown unit.
What unit do you want to convert to? inch
Sorry, unknown unit.
What unit do you want to convert to? grrr
  
```

To eliminate frustration, it is better to list the units that the user can supply.

```

From unit (in, ft, mi, mm, cm, m, km, oz, lb, g, kg, tsp, tbsp, pint, gall): cm
To unit: in ————— No need to list the units again
  
```

We switched to a shorter prompt to make room for all the unit names. Exercise R6.22 explores a different alternative.

There is another issue that we haven't addressed yet. How does the user quit the program? The first storyboard suggests that the program will go on forever.

We can ask the user after seeing the sentinel that terminates an input sequence.

Exiting the ProgramFrom unit (in, ft, mi, mm, cm, m, km, oz, lb, g, kg, tsp, tbsp, pint, gall): **cm**To unit: **in**

Enter values, terminated by zero

30**30 cm = 11.81 in****0**More conversions (y, n)? **n**

(Program exits)

Sentinel triggers the prompt to exit

As you can see from this case study, a storyboard is essential for developing a working program. You need to know the flow of the user interaction in order to structure your program.

SELF CHECK

26. Provide a storyboard panel for a program that reads a number of test scores and prints the average score. The program only needs to process one set of scores. Don't worry about error handling.
27. Google has a simple interface for converting units. You just type the question, and you get the answer.



Make storyboards for an equivalent interface in a Java program. Show a scenario in which all goes well, and show the handling of two kinds of errors.

28. Consider a modification of the program in Self Check 26. Suppose we want to drop the lowest score before computing the average. Provide a storyboard for the situation in which a user only provides one score.
29. What is the problem with implementing the following storyboard in Java?

Computing Multiple AveragesEnter scores: **90 80 90 100 80**The average is **88**Enter scores: **100 70 70 100 80**The average is **88**Enter scores: **-1**

(Program exits)

-1 is used as a sentinel to exit the program

30. Produce a storyboard for a program that compares the growth of a \$10,000 investment for a given number of years under two interest rates.

Practice It Now you can try these exercises at the end of the chapter: R6.21, R6.22, R6.23.

6.7 Common Loop Algorithms

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.

6.7.1 Sum and Average Value

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.

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

Note that the `total` variable is declared outside the loop. We want the loop to update a single variable. The `input` variable is declared inside the loop. A separate variable is created for each input and removed at the end of each loop iteration.

To compute an average, count how many values you have, and divide by the count. Be sure to check that the count is not zero.

```
double total = 0;
int count = 0;
while (in.hasNextDouble())
{
    double input = in.nextDouble();
    total = total + input;
    count++;
}
double average = 0;
if (count > 0)
{
    average = total / count;
}
```

To compute an average, keep a total and a count of all values.

6.7.2 Counting Matches

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

```
int spaces = 0;
for (int i = 0; i < str.length(); i++)
{
    char ch = str.charAt(i);
    if (ch == ' ')
    {
        spaces++;
    }
}
```

For example, if `str` is "My Fair Lady", `spaces` is incremented twice (when `i` is 2 and 7).

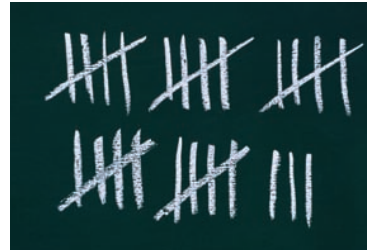
To count values that fulfill a condition, check all values and increment a counter for each match.

Note that the `spaces` variable is declared outside the loop. We want the loop to update a single variable. The `ch` variable is declared inside the loop. A separate variable is created for each iteration and removed at the end of each loop iteration.

This loop can also be used for scanning inputs. The following loop reads text a word at a time and counts the number of words with at most three letters:

```
int shortWords = 0;
while (in.hasNext())
{
    String input = in.next();
    if (input.length() <= 3)
    {
        shortWords++;
    }
}
```

In a loop that counts matches, a counter is incremented whenever a match is found.



6.7.3 Finding the First Match

If your goal is to find a match, exit the loop when the match is found.

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 space in a string. Because we do not visit all elements in the string, a `while` loop is a better choice than a `for` loop:

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

If a match was found, then `found` is `true`, `ch` is the first matching character, and `position` is the index of the first match. If the loop did not find a match, then `found` remains `false` after the end of the loop.

Note that the variable `ch` is declared *outside* the `while` loop because you may want to use the input after the loop has finished. If it had been declared inside the loop body, you would not be able to use it outside the loop.



When searching, you look at items until a match is found.

6.7.4 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:

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

Note that the variable `input` is declared *outside* the `while` loop because you will want to use the input after the loop has finished.

6.7.5 Maximum and Minimum

To find the largest value, update the largest value seen so far whenever you see a larger one.

To compute the largest value in a sequence, keep a variable that stores the largest element that you have encountered, and update it when you find a larger one.

```
double largest = in.nextDouble();
while (in.hasNextDouble())
{
    double input = in.nextDouble();
    if (input > largest)
    {
        largest = input;
    }
}
```

This algorithm requires that there is at least one input.

To compute the smallest value, simply reverse the comparison:

```
double smallest = in.nextDouble();
while (in.hasNextDouble())
{
    double input = in.nextDouble();
    if (input < smallest)
    {
        smallest = input;
    }
}
```

To find the height of the tallest bus rider, remember the largest value so far, and update it whenever you see a taller one.



6.7.6 Comparing Adjacent Values

To compare adjacent inputs, store the preceding input in a variable.

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, such as 1 7 2 9 4 9, contains adjacent duplicates.

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

```
double input;
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:

```
double input = 0;
while (in.hasNextDouble())
{
    double previous = input;
    input = in.nextDouble();
    if (input == previous)
    {
        System.out.println("Duplicate input");
    }
}
```



When comparing adjacent values, store the previous value in a variable.

One problem remains. When the loop is entered for the first time, `input` has not yet been read. You can solve this problem with an initial input operation outside the loop:

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



FULL CODE EXAMPLE

Go to wiley.com/go/javacode to download a program that uses common loop algorithms.

SELF CHECK



31. What total is computed when no user input is provided in the algorithm in Section 6.7.1?
32. How do you compute the total of all positive inputs?
33. What are the values of `position` and `ch` when no match is found in the algorithm in Section 6.7.3?
34. What is wrong with the following loop for finding the position of the first space in a string?

```
boolean found = false;
for (int position = 0; !found && position < str.length(); position++)
{
```

```

char ch = str.charAt(position);
if (ch == ' ') { found = true; }
}

```

35. How do you find the position of the *last* space in a string?
36. What happens with the algorithm in Section 6.7.6 when no input is provided at all? How can you overcome that problem?

Practice It Now you can try these exercises at the end of the chapter: E6.5, E6.9, E6.10.

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.

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

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

Read first value.

Read second value.

If second value is higher than the first, set highest temperature to that value, highest month to 2.

Read next value.

If value is higher than the first and second, set highest temperature to that value, highest month to 3.

Read next value.

If value is higher than the highest temperature seen so far, set highest temperature to that value, highest month to 4.

...

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:

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.