CHAPTER 4

LOOPS

CHAPTER GOALS

To implement `while`, `for`, and `do` loops

To hand-trace the execution of a program

To become familiar with common loop algorithms

To understand nested loops

To implement programs that read and process data sets

To use a computer for simulations

CHAPTER CONTENTS

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.

4.1 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?

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

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.
```

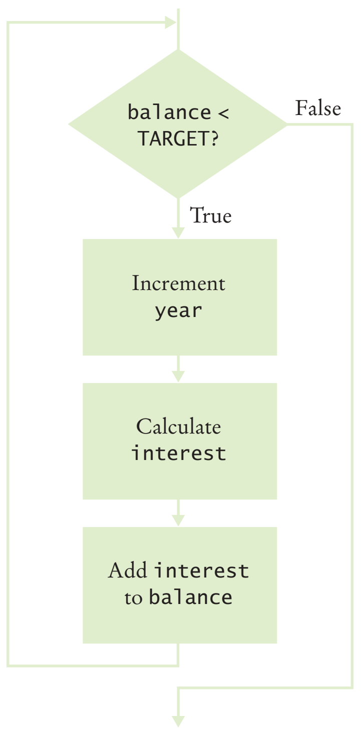
year	interest	balance
0		\$10,000

```
Repeat the following steps while the balance is less than $20,000.
    Add 1 to the year value.
    Compute the interest as balance x 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".

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 4.1). 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:

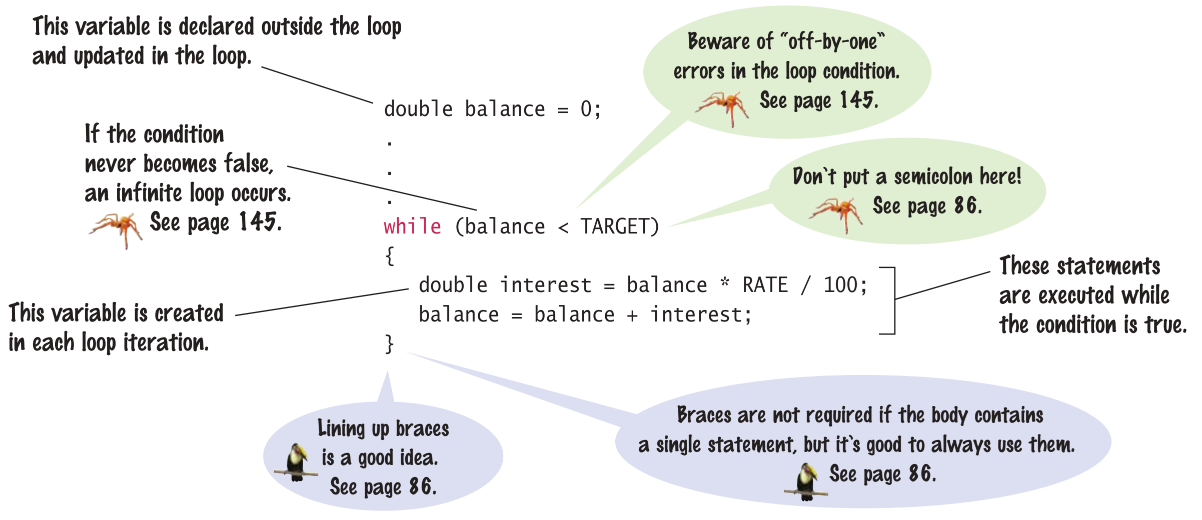
```
while (balance < TARGET)
{
   year++;
   double interest = balance * RATE / 100;
   balance = balance + interest;
}
```

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 4.1 while Statement

Syntax

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



```
double balance = 0;
.
.
.
while (balance < TARGET)
{
   double interest = balance * RATE / 100;
   balance = balance + interest;
}
```

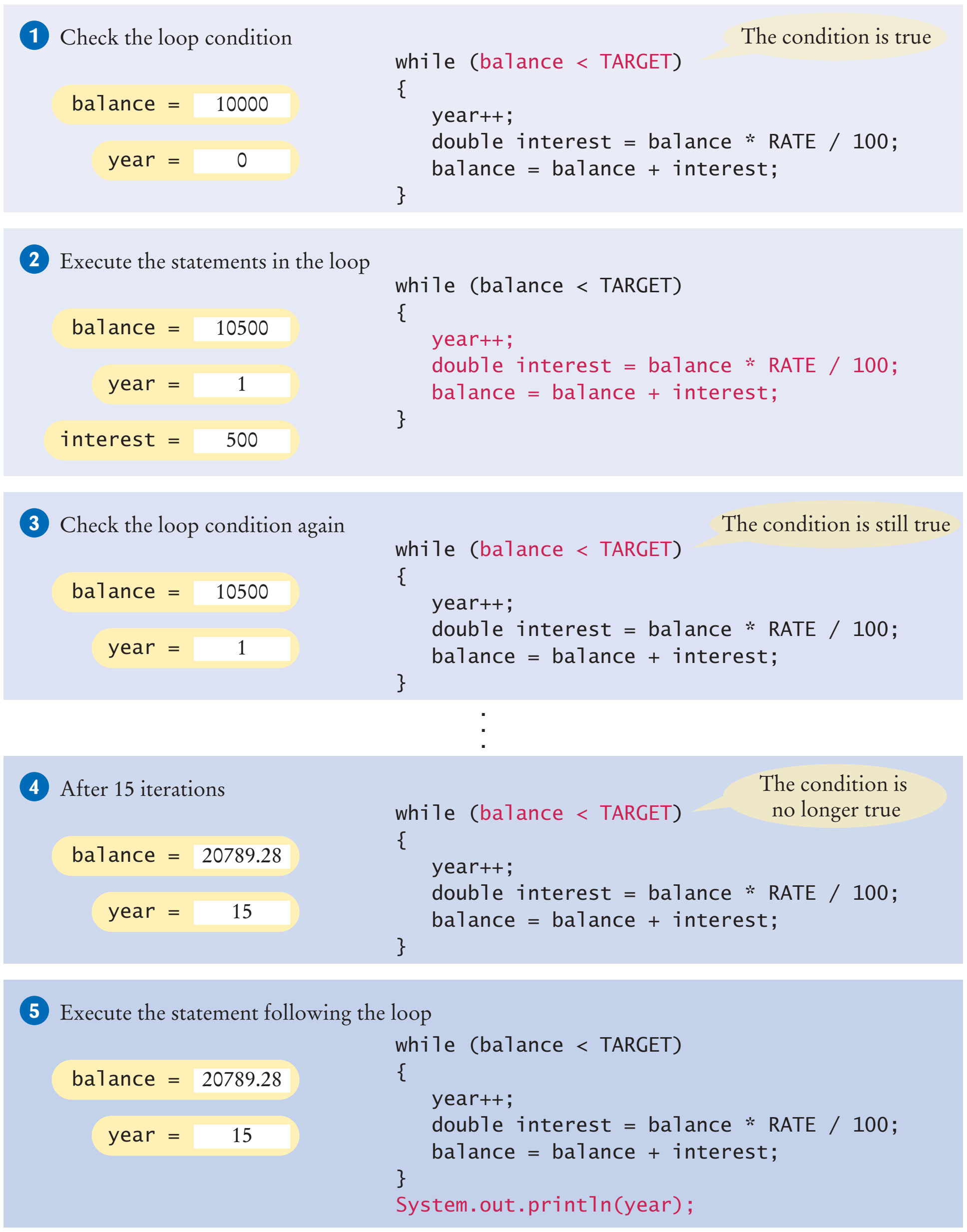
This variable is declared outside the loop and updated in the loop.
If the condition never becomes false, an infinite loop occurs. See page 145.
Beware of "off-by-one" errors in the loop condition. See page 145.
Don't put a semicolon here! See page 86.
These statements are executed while the condition is true.
This variable is created in each loop iteration.
Lining up braces is a good idea. See page 86.
Braces are not required if the body contains a single statement, but it's good to always use them. See page 86.

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

A new `interest` variable is created in each iteration.

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


1 Check the loop condition
The condition is true
balance = 10000
year = 0
2 Execute the statements in the loop
balance = 10500
year = 1
interest = 500
3 Check the loop condition again
The condition is still true
balance = 10500
year = 1
4 After 15 iterations
The condition is no longer true
balance = 20789.28
year = 15
5 Execute the statement following the loop
balance = 20789.28
year = 15

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

Figure 2
Execution of the `DoubleInvestment` Loop

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

section_1/DoubleInvestment.java

```java
/**
   This program computes the time required to double an investment.
*/
public class DoubleInvestment
{
   public static void main(String[] args)
   {
      final double RATE = 5;
      final double INITIAL_BALANCE = 10000;
      final double TARGET = 2 * INITIAL_BALANCE;

      double balance = INITIAL_BALANCE;
      int year = 0;

      // Count the years required for the investment to double

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

      System.out.println("The investment doubled after "
         + year + " years.");
   }
}
```

Program Run

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

SELF CHECK

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

   ```java
   while (balance <= TARGET)
   ```

   What is the effect on the program? Why?
5. What does the following loop print?

   ```java
   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: R4.1, R4.5, P4.14.

Table 1 while Loop Examples

Loop	Output	Explanation

`i = 0; sum = 0;`
`while (sum < 10)`
`{`
`   i++; sum = sum + i;`
`   Print i 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;`
`   Print i and sum;`
`}`
	`1 -1`
	`2 -3`
	`3 -6`
	`4 -10`
	`. . .`
		Because `sum` never reaches 10, this is an "infinite loop" (see Common Error 4.2 on page 145).

`i = 0; sum = 0;`
`while (sum < 0)`
`{`
`   i++; sum = sum - i;`
`   Print i 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;`
`   Print i 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 (see Common Error 4.1 on page 144).

`i = 0; sum = 0;`
`while (sum < 10) ;`
`{`
`   i++; sum = sum + i;`
`   Print i and sum;`
`}`
	(No output, program does not terminate)	Note the semicolon before the `{`. This loop has an empty body. It runs forever, checking whether `sum < 0` and doing nothing in the body.

Common Error 4.1

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 >= TARGET
```

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

```
while (balance < TARGET)
```

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 4.2

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:

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

```
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.)

Common Error 4.3

Off-by-One Errors

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

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

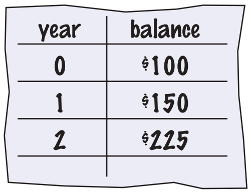
Should year start at 0 or at 1? Should you test for balance < TARGET or for balance <= TARGET? It is easy to be *off by one* in these expressions.

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.

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

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.



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 < TARGET`, the loop stops, as it should. If the test condition had been `balance <= TARGET`, the loop would have executed once more.

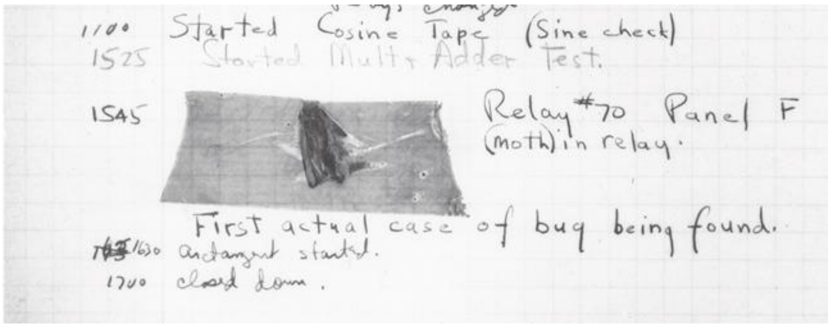
In other words, you keep adding interest while the balance *has not yet doubled*.

Random Fact 4.1 The First Bug

According to legend, the first bug was found 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 photo), it appears as if the term "bug" had already been in active use at the time.

The pioneering computer scientist Maurice Wilkes wrote, "Somehow, at the Moore School and afterwards, one had always assumed there would be no 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."



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

The First Bug

4.2 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.

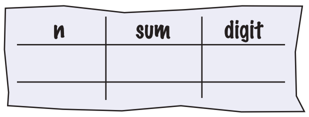
In Programming Tip 3.5, you learned about the method of hand-tracing. When you hand-trace code or pseudocode, you write the names of the variables on a sheet of paper, mentally execute each step of the code and update the variables.

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);
```

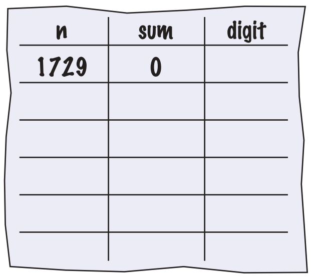
There are three variables: n, sum, and digit.



n	sum	digit

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

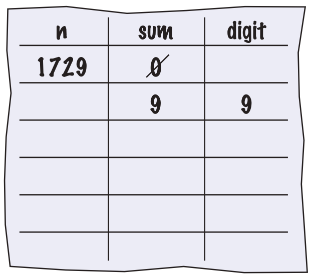
```
int n = 1729;
int sum = 0;
while (n > 0)
{
   int digit = n % 10;
   sum = sum + digit;
   n = n / 10;
}
System.out.println(sum);
```



n	sum	digit
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);
```



n	sum	digit
1729	~~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.

```
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
~~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.

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—see Exercise P4.32.)

Hand-tracing can show errors in code or pseudocode.

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: R4.3, R4.6.

4.3 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.

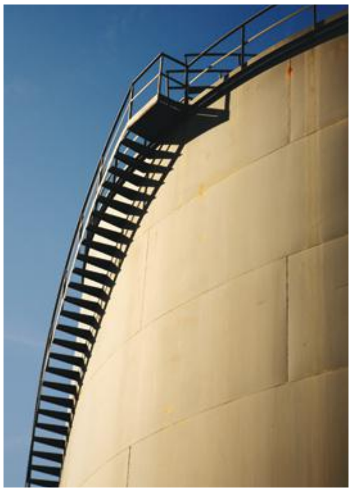
It often happens that you want to execute a sequence of statements a given number of times. You can use a `while` loop that is controlled by a counter, as in the following example:

```
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 4.2).

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

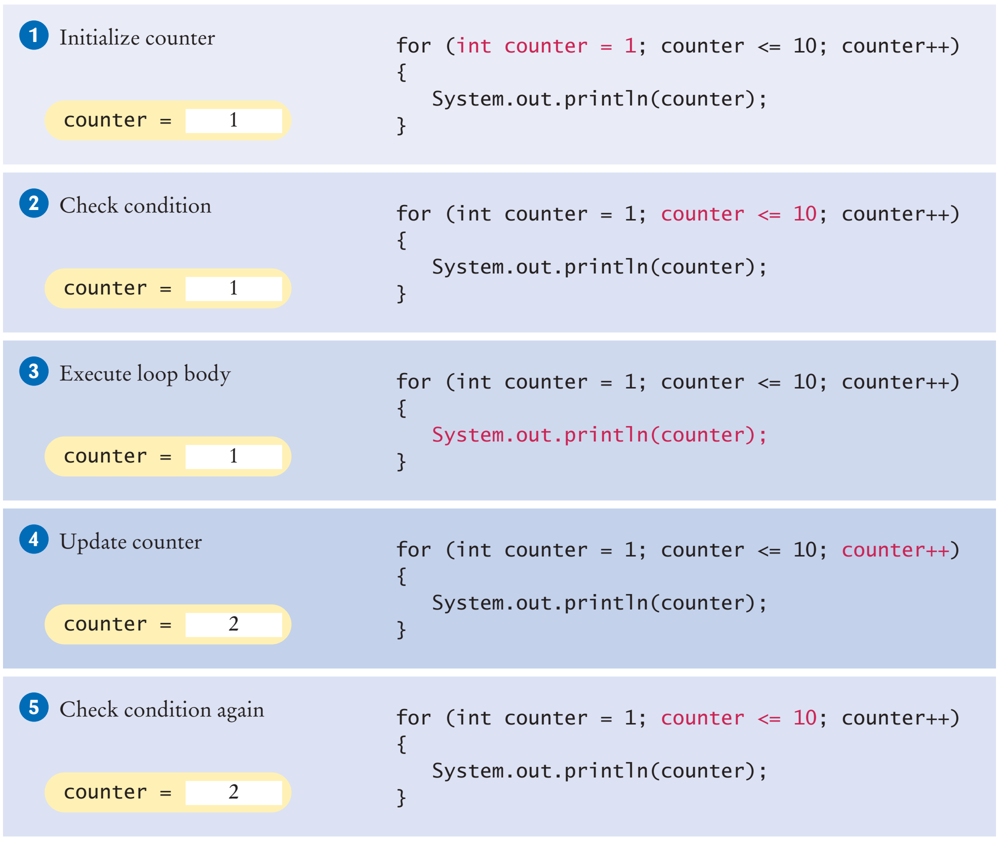
Some people call this loop *count-controlled*. In contrast, the `while` loop of the preceding section can be called an *event-controlled* loop because it executes until an event occurs; namely that the balance reaches the target. Another commonly used term for a count-controlled loop is *definite*. You 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.

ANIMATION
The `for` Loop

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).

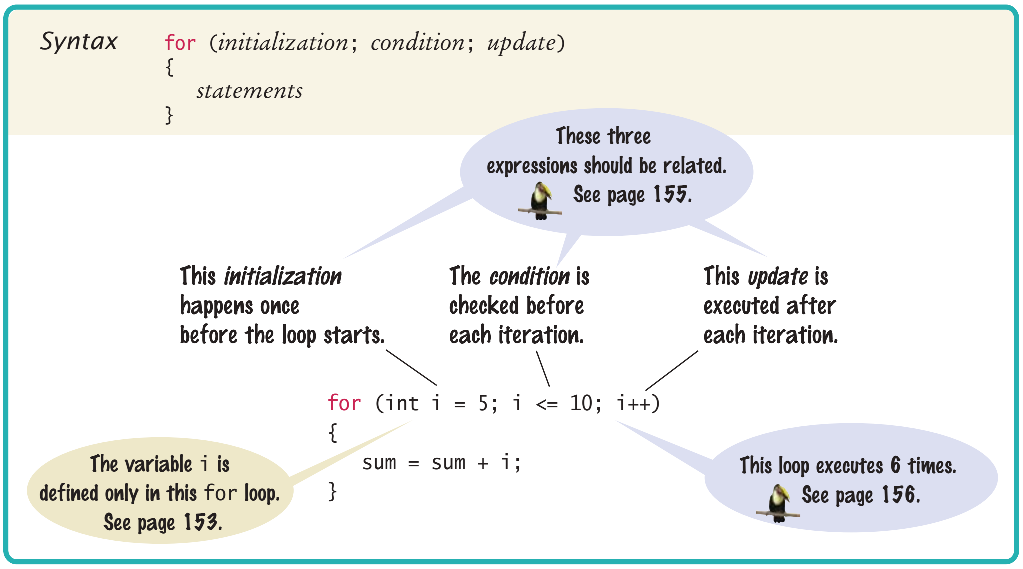
- The initialization is executed once, before the loop is entered. ①
- The condition is checked before each iteration. ② ⑤
- The update is executed after each iteration. ④



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

Figure 3 Execution of a `for` Loop

Syntax 4.2 for Statement



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

```
for (int i = 5; i <= 10; i++)
{
   sum = sum + i;
}
```

These three expressions should be related. See page 155.
This initialization happens once before the loop starts.
The condition is checked before each iteration.
This update is executed after each iteration.
The variable i is defined only in this for loop. See page 153.
This loop executes 6 times. See page 156.

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 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
```

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 4.3 on page 156.)
`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 `i`th character.

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
```

Here is a typical use of the for loop. We want to print the balance 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 <= nyears; year++)
{
   Update balance.
   Print year and balance.
}
```

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

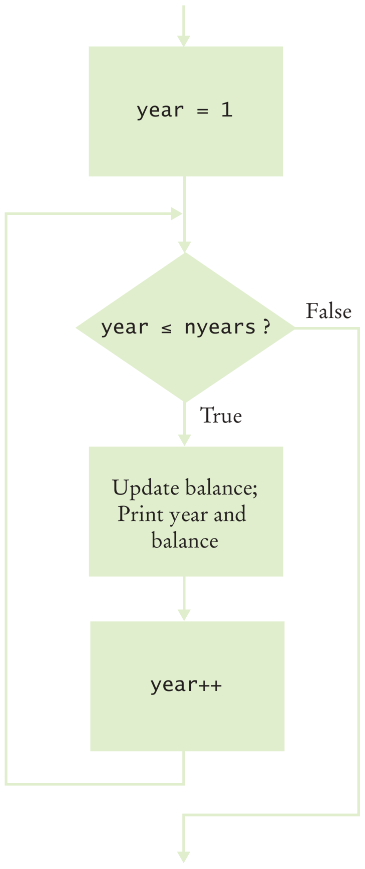


Figure 4 Flowchart of a for Loop

section_3/InvestmentTable.java

```
import java.util.Scanner;

/**
   This program prints a table showing the growth of an investment.
*/
public class InvestmentTable
{
   public static void main(String[] args)
   {
      final double RATE = 5;
      final double INITIAL_BALANCE = 10000;
      double balance = INITIAL_BALANCE;

      System.out.print("Enter number of years: ");
      Scanner in = new Scanner(System.in);
      int nyears = in.nextInt();

      // Print the table of balances for each year

      for (int year = 1; year <= nyears; year++)
      {
         double interest = balance * RATE / 100;
         balance = balance + interest;
         System.out.printf("%4d %10.2f\n", year, balance);
      }
   }
}
```

Program Run

```
Enter number of years: 10
   1  10500.00
   2  11025.00
   3  11576.25
   4  12155.06
   5  12762.82
   6  13400.96
   7  14071.00
   8  14774.55
   9  15513.28
  10  16288.95
```

Another 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.

11. Write the for loop of the InvestmentTable.java program 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 for loop of the InvestmentTable.java program to print all balances until the investment has doubled?

Practice It Now you can try these exercises at the end of the chapter: R4.4, R4.10, P4.8, P4.13.

Programming Tip 4.1

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 4.2

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 $1 \le i \le 10$. Because there are ≤ on both bounds, the bounds are called **symmetric**.

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 ith character as str.charAt(i). The values for i are bounded by 0 ≤ i < str.length(), with a ≤ to the left and a < to the right. That is appropriate, because str.length() is not a valid position. Such bounds are called **asymmetric**.

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 4.3

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.

4.4 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.

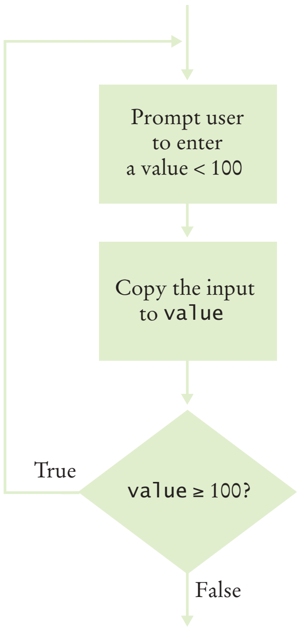
ONLINE EXAMPLE

A program to illustrate the use of the `do` loop for input validation.

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):

Figure 5 Flowchart of a do Loop

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



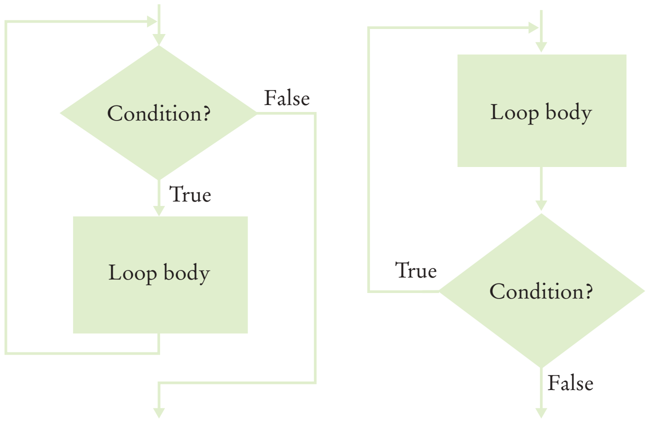
16. Suppose that we want to check for inputs that are at least 0 and at most 100. Modify the do loop for this check.
17. Rewrite the input check do loop using a while loop. What is the disadvantage of your solution?
18. Suppose Java didn't have a do loop. Could you rewrite any do loop as a while loop?
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: R4.9, R4.16, R4.17.

Programming Tip 4.4

Flowcharts for Loops

In Section 3.5, 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 3.5, 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.

4.5 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

```
import java.util.Scanner;

/**
   This program prints the average of salary values that are terminated with a sentinel.
*/
```

```
public class SentinelDemo
{
   public static void main(String[] args)
   {
      double sum = 0;
      int count = 0;
      double salary = 0;
      System.out.print("Enter salaries, -1 to finish: ");
      Scanner in = new Scanner(System.in);

      // Process data until the sentinel is entered

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

      // Compute and print the average

      if (count > 0)
      {
         double average = sum / count;
         System.out.println("Average salary: " + average);
      }
      else
      {
         System.out.println("No data");
      }
   }
}
```

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 4.1 on page 160 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 3.8, 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.
}
```

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: R4.13, P4.27, P4.28.

Special Topic 4.1

The Loop-and-a-Half Problem and the break Statement

Consider again this loop for processing inputs until a sentinel value has been reached:

```
boolean done = false;
while (!done)
{
   double value = in.nextDouble();
   if (value == -1)
   {
      done = true;
   }
   else
   {
      Process value.
   }
}
```

The actual 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.

As an alternative, you can use the break reserved word.

```
while (true)
{
   double value = in.nextDouble();
   if (value == -1) { break; }
   Process value.
}
```

The break statement breaks out of the enclosing loop, independent of the loop condition. When the break statement is encountered, the loop is terminated, and the statement following the loop is executed.

In the loop-and-a-half case, break statements can be beneficial. But it is difficult to lay down clear rules as to when they are safe and when they should be avoided. We do not use the break statement in this book.

Special Topic 4.2

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

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

```
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.

VIDEO EXAMPLE 4.1

Evaluating a Cell Phone Plan

In this Video Example, you will learn how to design a program that computes the cost of a cell phone plan from actual usage data.

Available online in WileyPLUS and at www.wiley.com/college/horstmann.

4.6 Problem Solving: Storyboards

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?

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

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.

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

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?

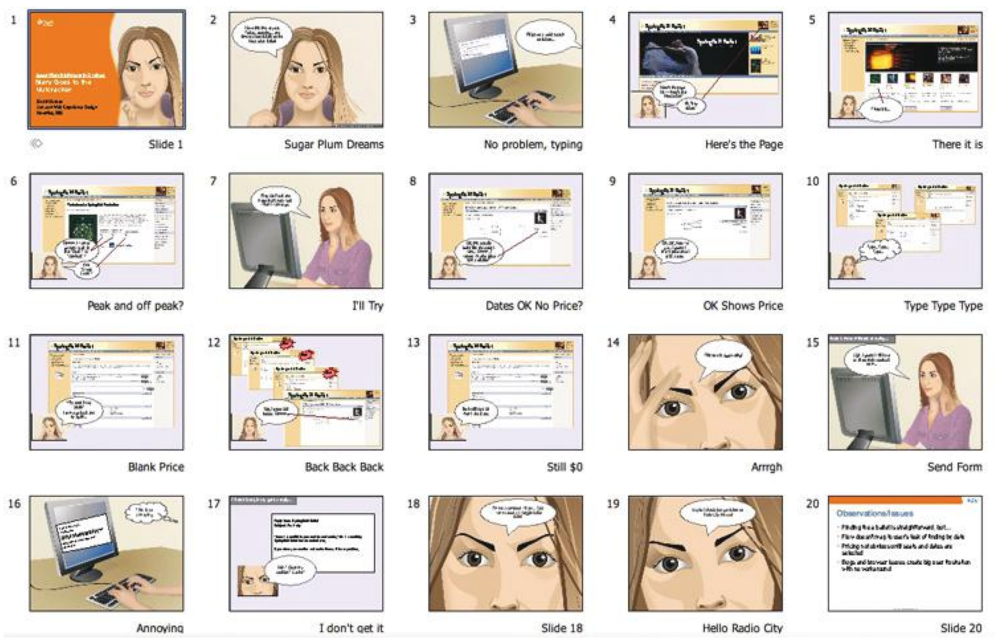


Figure 6 Storyboard for the Design of a Web Application

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, gal): 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 R4.21 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.

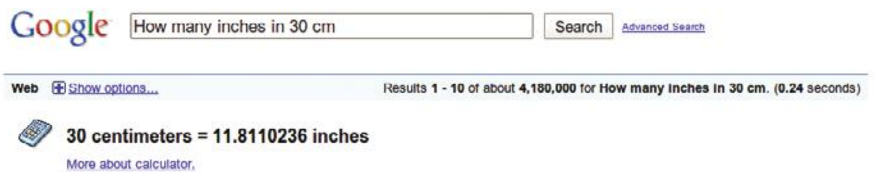
```
Exiting the Program

From unit (in, ft, mi, mm, cm, m, km, oz, lb, g, kg, tsp, tbsp, pint, gal): cm
To unit: in
Enter values, terminated by zero
30
30 cm = 11.81 in
0
More conversions (y, n)? n          Sentinel triggers the prompt to exit
(Program exits)
```

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.


Google
How many inches in 30 cm
Search
Advanced Search
Web
Show options...
Results 1 - 10 of about 4,180,000 for How many inches in 30 cm. (0.24 seconds)

30 centimeters = 11.8110236 inches

More about calculator.

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 Averages

Enter scores: 90 80 90 100 80
The average is 88
Enter scores: 100 70 70 100 80
The average is 88
Enter scores: -1          -1 is used as a sentinel to exit the program
(Program exits)
```

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: R4.21, R4.22, R4.23.

4.7 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.

4.7.1 Sum and Average Value

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

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;
}
```

4.7.2 Counting Matches

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

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).

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.

4.7.3 Finding the First Match

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

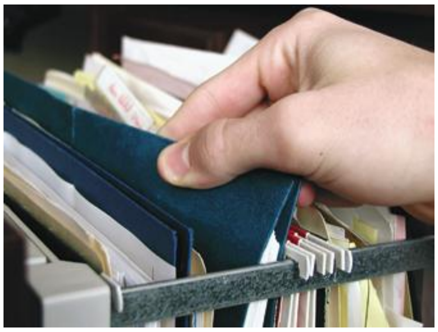
When you count the values that fulfill a condition, you need to look at all values. However, if your task is to find a match, then you can stop as soon as the condition is fulfilled.

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.

4.7.4 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.

4.7.5 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.

4.7.6 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 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:

```
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:

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

```
double input = 0;
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, `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");
   }
}
```

ONLINE EXAMPLE

A program using common loop algorithms.

SELF CHECK

31. What total is computed when no user input is provided in the algorithm in Section 4.7.1?
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 4.7.3?
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 4.7.6 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: P4.5, P4.9, P4.10.

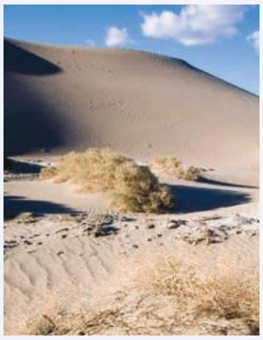
HOW TO 4.1 Writing a Loop

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, 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.

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:

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

Altogether our loop is

```
Repeat
    Read next value.
    If value is higher than the highest temperature,
        set highest temperature to that value,
        set 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 *count-controlled* loop is executed a definite number of times. In an *event-controlled* loop, the number of iterations is not known in advance—the loop is executed until some event happens.

Count-controlled loops can be implemented as `for` statements. For other loops, consider the loop condition. Do you need to complete one iteration of the loop body before you can tell when to terminate the loop? In that case, choose a `do` loop. 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:

```
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 body 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 read in only 11 more 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 to 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 from 2 to 12
    Read next value.
    If value is higher than the highest value
        Set highest value to that value.
        Set highest month to current month.
```

Step 6 Trace the loop with typical examples.

Hand trace your loop code, as described in Section 4.2. 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:

current month	current value	highest month	highest value
		~~1~~	~~22.6~~
~~2~~	~~36.6~~	~~2~~	~~36.6~~
~~3~~	~~44.5~~	3	44.5
4	24.2		

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 P4.4 asks you to complete the program.

```
double highestValue;
highestValue = in.nextDouble();
int highestMonth = 1;
```

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

WORKED EXAMPLE 4.1 **Credit Card Processing**

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

4.8 Nested Loops

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.

In Section 3.4, you saw how to nest two `if` statements. Similarly, complex iterations sometimes require a **nested loop**: a loop inside another loop statement. When processing tables, nested loops occur naturally. An outer loop iterates over all rows of the table. An inner loop deals with the columns in the current row.

In this section you will see how to print a table. For simplicity, we will simply print the powers of x, x^n, as in the table at right.

Here is the pseudocode for printing the table:

```
Print table header.
For x from 1 to 10
   Print table row.
   Print new line.
```

x^1	x^2	x^3	x^4
1	1	1	1
2	4	8	16
3	9	27	81
…	…	…	…
10	100	1000	10000

How do you print a table row? You need to print a value for each exponent. This requires a second loop.

```
For n from 1 to 4
   Print $x^n$.
```

This loop must be placed inside the preceding loop. We say that the inner loop is *nested* inside the outer loop.

The hour and minute displays in a digital clock are an example of nested loops. The hours loop 12 times, and for each hour, the minutes loop 60 times.

Available online in WileyPLUS and at `www.wiley.com/college/horstmann`.

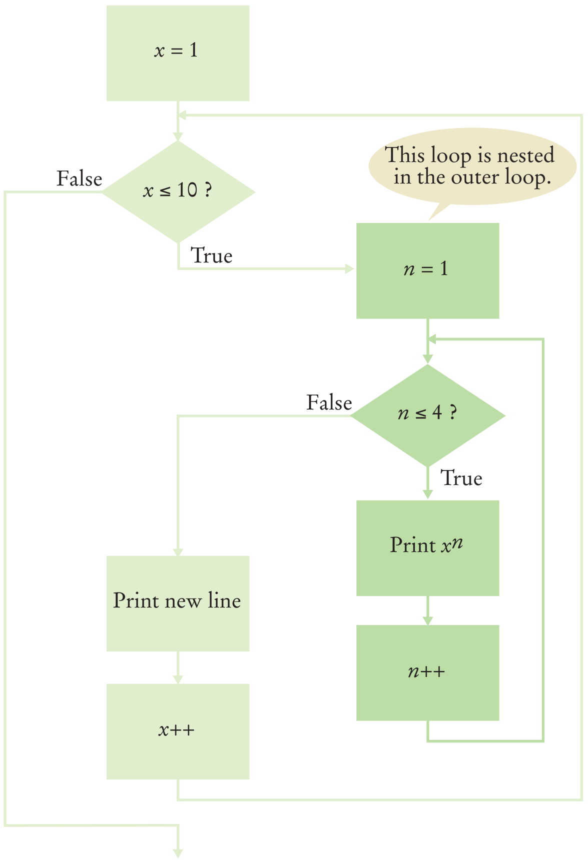


Figure 7
Flowchart of a Nested Loop

There are 10 rows in the outer loop. For each x, the program prints four columns in the inner loop (see Figure 7). Thus, a total of $10 \times 4 = 40$ values are printed.

Following is the complete program. Note that we also use loops to print the table header. However, those loops are not nested.

section_8/PowerTable.java

```
/**
   This program prints a table of powers of x.
*/
public class PowerTable
{
   public static void main(String[] args)
   {
      final int NMAX = 4;
      final double XMAX = 10;

      // Print table header

      for (int n = 1; n <= NMAX; n++)
      {
         System.out.printf("%10d", n);
      }
      System.out.println();
      for (int n = 1; n <= NMAX; n++)
      {
         System.out.printf("%10s", "x ");
      }
      System.out.println();

      // Print table body

      for (double x = 1; x <= XMAX; x++)
      {
         // Print table row

         for (int n = 1; n <= NMAX; n++)
         {
            System.out.printf("%10.0f", Math.pow(x, n));
         }
         System.out.println();
      }
   }
}
```

Program Run

```
         1         2         3         4
        x         x         x         x 

         1         1         1         1
         2         4         8        16
         3         9        27        81
         4        16        64       256
         5        25       125       625
         6        36       216      1296
         7        49       343      2401
         8        64       512      4096
         9        81       729      6561
        10       100      1000     10000
```

SELF CHECK

37. Why is there a statement `System.out.println();` in the outer loop but not in the inner loop?

38. How would you change the program to display all powers from x^0 to x^5?

39. If you make the change in Self Check 38, how many values are displayed?

40. What do the following nested loops display?

```
for (int i = 0; i < 3; i++)
{
   for (int j = 0; j < 4; j++)
   {
      System.out.print(i + j);
   }
   System.out.println();
}
```

41. Write nested loops that make the following pattern of brackets:

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

Practice It Now you can try these exercises at the end of the chapter: R4.27, P4.19, P4.21.

Table 3 Nested Loop Examples

Nested Loops	Output	Explanation

`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.

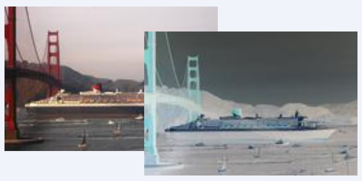
`for (i = 1; i <= 4; i++)`
`{`
`   for (j = 1; j <= i; j++) { Print "*" }`
`   System.out.println();`
`}`
	`*`
	`**`
	`***`
	`****`
		Prints 4 rows of lengths 1, 2, 3, and 4.

`for (i = 1; i <= 3; i++)`
`{`
`   for (j = 1; j <= 5; j++)`
`   {`
`      if (j % 2 == 0) { Print "*" }`
`      else { Print "-" }`
`   }`
`   System.out.println();`
`}`
	`-*-*-`
	`-*-*-`
	`-*-*-`
		Prints asterisks in even columns, dashes in odd columns.

`for (i = 1; i <= 3; i++)`
`{`
`   for (j = 1; j <= 5; j++)`
`   {`
`      if (i % 2 == j % 2) { Print "*" }`
`      else { Print " " }`
`   }`
`   System.out.println();`
`}`
	`* * *`
	` * * `
	`* * *`
		Prints a checkerboard pattern.

WORKED EXAMPLE 4.2 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.



⊕ Available online in WileyPLUS and at www.wiley.com/college/horstmann.

4.9 Application: Random Numbers and Simulations

In a simulation, you use the computer to simulate an activity.

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. You will see examples in the following sections.

4.9.1 Generating Random Numbers

Many events in the real world are difficult to predict with absolute precision, yet we can sometimes know the average behavior quite well. For example, a store may know from experience that a customer arrives every five minutes. Of course, that is an average—customers don't arrive in five minute intervals. To accurately model customer traffic, you want to take that random fluctuation into account. Now, how can you run such a simulation in the computer?

You can introduce randomness by calling the random number generator.

The Java library has a *random number generator*, which produces numbers that appear to be completely random. Calling `Math.random()` yields a random floating-point number that is ≥ 0 and < 1. Call `Math.random()` again, and you get a different number.

The following program calls `Math.random()` ten times.

section_9_1/RandomDemo.java

```
/**
   This program prints ten random numbers between 0 and 1.
*/
public class RandomDemo
{
   public static void main(String[] args)
   {
      for (int i = 1; i <= 10; i++)
      {
         double r = Math.random();
         System.out.println(r);
      }
   }
}
```

Program Run

```
0.6513550469421886
0.920193662882893
0.6904776061289993
0.8862828776788884
0.7730177555323139
0.3020238718668635
0.0028504531690907164
0.9099983981705169
0.1151636530517488
0.1592258808929058
```

Actually, the numbers are not completely random. They are drawn from sequences of numbers that don't repeat for a long time. These sequences are actually computed from fairly simple formulas; they just behave like random numbers (see Exercise P4.25). For that reason, they are often called **pseudorandom** numbers.

4.9.2 Simulating Die Tosses

In actual applications, you need to transform the output from the random number generator into different ranges. For example, to simulate the throw of a die, you need random integers between 1 and 6.

Here is the general recipe for computing random integers between two bounds a and b. As you know from Programming Tip 4.3 on page 156, there are b - a + 1 values between a and b, including the bounds themselves. First compute

```
(int) (Math.random() * (b - a + 1))
```

to obtain a random integer between 0 and b - a, then add a, yielding a random value between a and b:

```
int r = (int) (Math.random() * (b - a + 1)) + a;
```

Here is a program that simulates the throw of a pair of dice:

section_9_2/Dice.java

```
/**
   This program simulates tosses of a pair of dice.
*/
public class Dice
{
   public static void main(String[] args)
   {
      for (int i = 1; i <= 10; i++)
      {
         // Generate two random numbers between 1 and 6

         int d1 = (int) (Math.random() * 6) + 1;
         int d2 = (int) (Math.random() * 6) + 1;
         System.out.println(d1 + " " + d2);
      }
      System.out.println();
   }
}
```

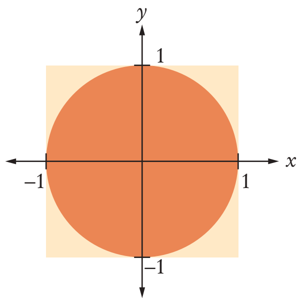
Program Run

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

4.9.3 The Monte Carlo Method

The Monte Carlo method is an ingenious method for finding approximate solutions to problems that cannot be precisely solved. (The method is named after the famous casino in Monte Carlo.) Here is a typical example. It is difficult to compute the number π, but you can approximate it quite well with the following simulation.

Simulate shooting a dart into a square surrounding a circle of radius 1. That is easy: generate random x and y coordinates between –1 and 1.



If the generated point lies inside the circle, we count it as a *hit*. That is the case when $x^2 + y^2 \le 1$. Because our shots are entirely random, we expect that the ratio of *hits / tries* is approximately equal to the ratio of the areas of the circle and the square, that is, π / 4. Therefore, our estimate for π is 4 × *hits / tries*. This method yields an estimate for π, using nothing but simple arithmetic.

To generate a random floating-point value between –1 and 1, you compute:

```
double r = Math.random(); // 0 ≤ r < 1
double x = -1 + 2 * r; // –1 ≤ x < 1
```

As r ranges from 0 (inclusive) to 1 (exclusive), x ranges from –1 + 2 × 0 = –1 (inclusive) to –1 + 2 × 1 = 1 (exclusive). In our application, it does not matter that x never reaches 1. The points that fulfill the equation $x = 1$ lie on a line with area 0.

Here is the program that carries out the simulation:

section_9_3/MonteCarlo.java

```
/**
   This program computes an estimate of pi by simulating dart throws onto a square.
*/
public class MonteCarlo
{
   public static void main(String[] args)
   {
      final int TRIES = 10000;

      int hits = 0;
      for (int i = 1; i <= TRIES; i++)
      {
         // Generate two random numbers between -1 and 1

         double r = Math.random();
         double x = -1 + 2 * r; // Between -1 and 1
         r = Math.random();
         double y = -1 + 2 * r;

         // Check whether the point lies in the unit circle

         if (x * x + y * y <= 1) { hits++; }
      }

      /*
         The ratio hits / tries is approximately the same as the ratio
         circle area / square area = pi / 4
      */

      double piEstimate = 4.0 * hits / TRIES;
      System.out.println("Estimate for pi: " + piEstimate);
   }
}
```

Program Run

```
Estimate for pi: 3.1504
```

SELF CHECK

42. How do you simulate a coin toss with the `Math.random()` method?

43. How do you simulate the picking of a random playing card?

44. Why does the loop body in `Dice.java` call `Math.random()` twice?

45. In many games, you throw a pair of dice to get a value between 2 and 12. What is wrong with this simulated throw of a pair of dice?

```
int sum = (int) (Math.random() * 11) + 2;
```

46. How do you generate a random floating-point number ≥ 0 and < 100?

Practice It Now you can try these exercises at the end of the chapter: R4.28, P4.7, P4.24.

Special Topic 4.3

Drawing Graphical Shapes

In Java, it is easy to produce simple drawings such as the one in Figure 8. By writing programs that draw such patterns, you can practice programming loops. For now, we give you a program outline into which you place your drawing code. The program outline also contains the necessary code for displaying a window containing your drawing. You need not look at that code now. It will be discussed in detail in Chapter 10.

Your drawing instructions go inside the `draw` method:

```
public class TwoRowsOfSquares
{
   public static void draw(Graphics g)
   {
      Drawing instructions
   }
   . . .
}
```

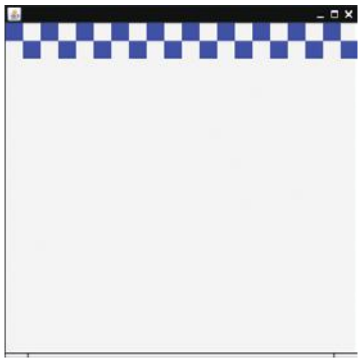


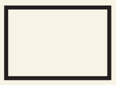
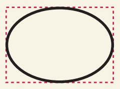
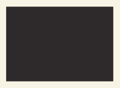
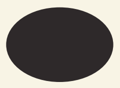
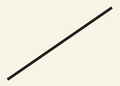
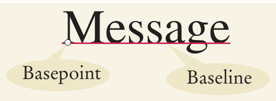
Figure 8 Two Rows of Squares

When the window is shown, the `draw` method is called, and your drawing instructions will be executed.

The draw method receives an object of type Graphics. The Graphics object has methods for drawing shapes. It also remembers the color that is used for drawing operations. You can think of the Graphics object as the equivalent of System.out for drawing shapes instead of printing values.

Table 4 shows useful methods of the Graphics class.

Table 4 Graphics Methods

Method	Result	Notes
`g.drawRect(x, y, width, height)`		(x, y) is the top left corner.
`g.drawOval(x, y, width, height)`		(x, y) is the top left corner of the box that bounds the ellipse. To draw a circle, use the same value for width and height.
`g.fillRect(x, y, width, height)`		The rectangle is filled in.
`g.fillOval(x, y, width, height)`		The oval is filled in.
`g.drawLine(x1, y1, x2, y2)`		(x1, y1) and (x2, y2) are the endpoints.
`g.drawString("Message", x, y)`		(x, y) is the basepoint.
`g.setColor(color)`	From now on, draw or fill methods will use this color.	Use Color.RED, Color.GREEN, Color.BLUE, and so on. (See Table 10.1 for a complete list of predefined colors.)

The program below draws the squares shown in Figure 8. When you want to produce your own drawings, make a copy of this program and modify it. Replace the drawing tasks in the draw method. Rename the class (for example, Spiral instead of TwoRowsOfSquares).

special_topic_3/TwoRowsOfSquares.java

```java
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JFrame;
import javax.swing.JComponent;

/**
   This program draws two rows of squares.
*/
public class TwoRowsOfSquares
{
   public static void draw(Graphics g)
   {
      final int width = 20;
      g.setColor(Color.BLUE);

      // Top row. Note that the top left corner of the drawing has coordinates (0, 0)
      int x = 0;
      int y = 0;
      for (int i = 0; i < 10; i++)
      {
         g.fillRect(x, y, width, width);
         x = x + 2 * width;
      }
      // Second row, offset from the first one
      x = width;
      y = width;
      for (int i = 0; i < 10; i++)
      {
         g.fillRect(x, y, width, width);
         x = x + 2 * width;
      }
   }

   public static void main(String[] args)
   {
      // Do not look at the code in the main method
      // Your code will go into the draw method above

      JFrame frame = new JFrame();

      final int FRAME_WIDTH = 400;
      final int FRAME_HEIGHT = 400;

      frame.setSize(FRAME_WIDTH, FRAME_HEIGHT);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      JComponent component = new JComponent()
      {
         public void paintComponent(Graphics graph)
         {
            draw(graph);
         }
      };

      frame.add(component);
      frame.setVisible(true);
   }
}
```

VIDEO EXAMPLE 4.2 **Drawing a Spiral**

In this Video Example, you will see how to develop a program that draws a spiral.

Available online in WileyPLUS and at www.wiley.com/college/horstmann.

Random Fact 4.2 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 software 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.

CHAPTER SUMMARY

Explain the flow of execution in a loop.

- A loop executes instructions repeatedly while a condition is true.
- An off-by-one error is a common error when programming loops. Think through simple test cases to avoid this type of error.

Use the technique of hand-tracing to analyze the behavior of a program.

- Hand-tracing is a simulation of code execution in which you step through instructions and track the values of the variables.
- Hand-tracing can help you understand how an unfamiliar algorithm works.
- Hand-tracing can show errors in code or pseudocode.

Use `for` loops for implementing count-controlled loops.

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

Choose between the `while` loop and the `do` loop.

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

Implement loops that read sequences of input data.

- A sentinel value denotes the end of a data set, but it is not part of the data.
- You can use a Boolean variable to control a loop. Set the variable to `true` before entering the loop, then set it to `false` to leave the loop.
- Use input redirection to read input from a file. Use output redirection to capture program output in a file.

Use the technique of storyboarding for planning user interactions.

- 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.

Know the most common loop algorithms.

- To compute an average, keep a total and a count of all values.
- To count values that fulfill a condition, check all values and increment a counter for each match.
- If your goal is to find a match, exit the loop when the match is found.
- To find the largest value, update the largest value seen so far whenever you see a larger one.
- To compare adjacent inputs, store the preceding input in a variable.

Use nested loops to implement multiple levels of iteration.

- 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.

- In a simulation, you use the computer to simulate an activity.
- You can introduce randomness by calling the random number generator.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

```
java.awt.Color              java.lang.Math
java.awt.Graphics              random
   drawLine
   drawOval
   drawRect
   drawString
   setColor
```

REVIEW EXERCISES

■ **R4.1** Write a `while` loop that prints

a. All squares less than `n`. For example, if `n` is 100, print 0 1 4 9 16 25 36 49 64 81.

b. All positive numbers that are divisible by 10 and less than `n`. For example, if `n` is 100, print 10 20 30 40 50 60 70 80 90

c. All powers of two less than `n`. For example, if `n` is 100, print 1 2 4 8 16 32 64.

■■ **R4.2** Write a loop that computes

a. The sum of all even numbers between 2 and 100 (inclusive).

b. The sum of all squares between 1 and 100 (inclusive).

c. The sum of all odd numbers between `a` and `b` (inclusive).

d. The sum of all odd digits of `n`. (For example, if `n` is 32677, the sum would be 3 + 7 + 7 = 17.)

■ **R4.3** Provide trace tables for these loops.

a.
```
int i = 0; int j = 10; int n = 0;
while (i < j) { i++; j--; n++; }
```
b.
```
int i = 0; int j = 0; int n = 0;
while (i < 10) { i++; n = n + i + j; j++; }
```
c.
```
int i = 10; int j = 0; int n = 0;
while (i > 0) { i--; j++; n = n + i - j; }
```
d.
```
int i = 0; int j = 10; int n = 0;
while (i != j) { i = i + 2; j = j - 2; n++; }
```

■ **R4.4** What do these loops print?

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

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

c. `for (int i = 10; i > 1; i--) { System.out.print(i + " "); }`

d. `for (int i = 0; i < 10; i++) { System.out.print(i + " "); }`

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

f. `for (int i = 1; i < 10; i++) { if (i % 2 == 0) { System.out.print(i + " "); } }`

■ **R4.5** What is an infinite loop? On your computer, how can you terminate a program that executes an infinite loop?

■ **R4.6** Write a program trace for the pseudocode in Exercise P4.6, assuming the input values are 4 7 –2 –5 0.

■■ **R4.7** What is an “off-by-one” error? Give an example from your own programming experience.

■ **R4.8** What is a sentinel value? Give a simple rule when it is appropriate to use a numeric sentinel value.

■ **R4.9** Which loop statements does Java support? Give simple rules for when to use each loop type.

■ **R4.10** How many iterations do the following loops carry out? Assume that `i` is not changed in the loop body.

a. `for (int i = 1; i <= 10; i++) . . .`
b. `for (int i = 0; i < 10; i++) . . .`
c. `for (int i = 10; i > 0; i--) . . .`
d. `for (int i = -10; i <= 10; i++) . . .`
e. `for (int i = 10; i >= 0; i++) . . .`
f. `for (int i = -10; i <= 10; i = i + 2) . . .`
g. `for (int i = -10; i <= 10; i = i + 3) . . .`

■■ **R4.11** Write pseudocode for a program that prints a calendar such as the following:

```
Su  M  T  W Th  F Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

■ **R4.12** Write pseudocode for a program that prints a Celsius/Fahrenheit conversion table such as the following:

```
Celsius | Fahrenheit
--------+-----------
      0 |         32
     10 |         50
     20 |         68
  . . .        . . .
    100 |        212
```

■ **R4.13** Write pseudocode for a program that reads a student record, consisting of the student’s first and last name, followed by a sequence of test scores and a sentinel of –1. The program should print the student’s average score. Then provide a trace table for this sample input:

```
Harry Morgan 94 71 86 95 -1
```

■■ **R4.14** Write pseudocode for a program that reads a sequence of student records and prints the total score for each student. Each record has the student’s first and last name, followed by a sequence of test scores and a sentinel of –1. The sequence is terminated by the word `END`. Here is a sample sequence:

```
Harry Morgan 94 71 86 95 -1
Sally Lin 99 98 100 95 90 -1
END
```

Provide a trace table for this sample input.

▪ **R4.15** Rewrite the following `for` loop into a `while` loop.

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

▪ **R4.16** Rewrite the following `do` loop into a `while` loop.

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

▪ **R4.17** Provide trace tables of the following loops.

a.
```
int s = 1;
int n = 1;
while (s < 10) { s = s + n; }
n++;
```

b.
```
int s = 1;
for (int n = 1; n < 5; n++) { s = s + n; }
```

c.
```
int s = 1;
int n = 1;
do
{
   s = s + n;
   n++;
}
while (s < 10 * n);
```

▪ **R4.18** What do the following loops print? Work out the answer by tracing the code, not by using the computer.

a.
```
int s = 1;
for (int n = 1; n <= 5; n++)
{
   s = s + n;
   System.out.print(s + " ");
}
```

b.
```
int s = 1;
for (int n = 1; s <= 10; System.out.print(s + " "))
{
   n = n + 2;
   s = s + n;
}
```

c.
```
int s = 1;
int n;
for (n = 1; n <= 5; n++)
{
   s = s + n;
   n++;
}
System.out.print(s + " " + n);
```

■ **R4.19** What do the following program segments print? Find the answers by tracing the code, not by using the computer.

a.
```
int n = 1;
for (int i = 2; i < 5; i++) { n = n + i; }
System.out.print(n);
```

b.
```
int i;
double n = 1 / 2;
for (i = 2; i <= 5; i++) { n = n + 1.0 / i; }
System.out.print(i);
```

c.
```
double x = 1;
double y = 1;
int i = 0;
do
{
   y = y / 2;
   x = x + y;
   i++;
}
while (x < 1.8);
System.out.print(i);
```

d.
```
double x = 1;
double y = 1;
int i = 0;
while (y >= 1.5)
{
   x = x / 2;
   y = x + y;
   i++;
}
System.out.print(i);
```

■■ **R4.20** Give an example of a `for` loop where symmetric bounds are more natural. Give an example of a `for` loop where asymmetric bounds are more natural.

■ **R4.21** Add a storyboard panel for the conversion program in Section 4.6 on page 162 that shows a scenario where a user enters incompatible units.

■ **R4.22** In Section 4.6, we decided to show users a list of all valid units in the prompt. If the program supports many more units, this approach is unworkable. Give a storyboard panel that illustrates an alternate approach: If the user enters an unknown unit, a list of all known units is shown.

■ **R4.23** Change the storyboards in Section 4.6 to support a menu that asks users whether they want to convert units, see program help, or quit the program. The menu should be displayed at the beginning of the program, when a sequence of values has been converted, and when an error is displayed.

■ **R4.24** Draw a flow chart for a program that carries out unit conversions as described in Section 4.6.

■■ **R4.25** In Section 4.7.5, the code for finding the largest and smallest input initializes the `largest` and `smallest` variables with an input value. Why can't you initialize them with zero?

■ **R4.26** What are nested loops? Give an example where a nested loop is typically used.

■■ **R4.27** The nested loops

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

display a rectangle of a given width and height, such as

```
****
****
****
```

Write a *single* `for` loop that displays the same rectangle.

■■ **R4.28** Suppose you design an educational game to teach children how to read a clock. How do you generate random values for the hours and minutes?

■■■ **R4.29** In a travel simulation, Harry will visit one of his friends that are located in three states. He has ten friends in California, three in Nevada, and two in Utah. How do you produce a random number between 1 and 3, denoting the destination state, with a probability that is proportional to the number of friends in each state?

PROGRAMMING EXERCISES

■ **P4.1** Write programs with loops that compute

- **a.** The sum of all even numbers between 2 and 100 (inclusive).
- **b.** The sum of all squares between 1 and 100 (inclusive).
- **c.** All powers of 2 from 2^0 up to 2^{20}.
- **d.** The sum of all odd numbers between `a` and `b` (inclusive), where `a` and `b` are inputs.
- **e.** The sum of all odd digits of an input. (For example, if the input is 32677, the sum would be 3 + 7 + 7 = 17.)

■■ **P4.2** Write programs that read a sequence of integer inputs and print

- **a.** The smallest and largest of the inputs.
- **b.** The number of even and odd inputs.
- **c.** Cumulative totals. For example, if the input is 1 7 2 9, the program should print 1 8 10 19.
- **d.** All adjacent duplicates. For example, if the input is 1 3 3 4 5 5 6 6 6 2, the program should print 3 5 6.

■■ **P4.3** Write programs that read a line of input as a string and print

- **a.** Only the uppercase letters in the string.
- **b.** Every second letter of the string.
- **c.** The string, with all vowels replaced by an underscore.
- **d.** The number of vowels in the string.
- **e.** The positions of all vowels in the string.

■■ **P4.4** Complete the program in How To 4.1 on page 169. Your program should read twelve temperature values and print the month with the highest temperature.

■■ **P4.5** Write a program that reads a set of floating-point values. Ask the user to enter the values, then print

- the average of the values.
- the smallest of the values.
- the largest of the values.
- the range, that is the difference between the smallest and largest.

Of course, you may only prompt for the values once.

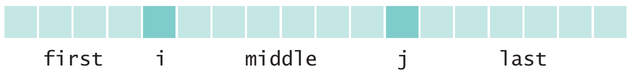
■ **P4.6** Translate the following pseudocode for finding the minimum value from a set of inputs into a Java program.

```
Set a Boolean variable "first" to true.
While another value has been read successfully
    If first is true
        Set the minimum to the value.
        Set first to false.
    Else if the value is less than the minimum
        Set the minimum to the value.
Print the minimum.
```

■■■ **P4.7** Translate the following pseudocode for randomly permuting the characters in a string into a Java program.

```
Read a word.
Repeat word.length() times
    Pick a random position i in the word, but not the last position.
    Pick a random position j > i in the word.
    Swap the letters at positions j and i.
Print the word.
```

To swap the letters, construct substrings as follows:



Then replace the string with

```
first + word.charAt(j) + middle + word.charAt(i) + last
```

■ **P4.8** Write a program that reads a word and prints each character of the word on a separate line. For example, if the user provides the input `"Harry"`, the program prints

```
H
a
r
r
y
```

■■ **P4.9** Write a program that reads a word and prints the word in reverse. For example, if the user provides the input `"Harry"`, the program prints

```
yrraH
```

■ **P4.10** Write a program that reads a word and prints the number of vowels in the word. For this exercise, assume that `a e i o u y` are vowels. For example, if the user provides the input `"Harry"`, the program prints `2 vowels`.

■■■ **P4.11** Write a program that reads a word and prints the number of syllables in the word. For this exercise, assume that syllables are determined as follows: Each sequence of adjacent vowels `a e i o u y`, except for the last `e` in a word, is a syllable. However, if that algorithm yields a count of `0`, change it to 1. For example,

Word	Syllables
`Harry`	2
`hairy`	2
`hare`	1
`the`	1

■■■ **P4.12** Write a program that reads a word and prints all substrings, sorted by length. For example, if the user provides the input `"rum"`, the program prints

```
r
u
m
ru
um
rum
```

■ **P4.13** Write a program that prints all powers of 2 from 2^0 up to 2^{20}.

■■ **P4.14** Write a program that reads a number and prints all of its *binary digits:* Print the remainder `number % 2`, then replace the number with `number / 2`. Keep going until the number is `0`. For example, if the user provides the input 13, the output should be

```
1
0
1
1
```

■■ **P4.15** *Mean and standard deviation*. Write a program that reads a set of floating-point data values. Choose an appropriate mechanism for prompting for the end of the data set. When all values have been read, print out the count of the values, the average, and the standard deviation. The average of a data set $\{x_1, \ldots, x_n\}$ is $\bar{x} = \sum x_i / n$, where $\sum x_i = x_1 + \ldots + x_n$ is the sum of the input values. The standard deviation is

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

However, this formula is not suitable for the task. By the time the program has computed $\bar{x}$, the individual x_i are long gone. Until you know how to save these values, use the numerically less stable formula

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

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

■■ **P4.16** The *Fibonacci numbers* are defined by the sequence

$$f_1 = 1$$
$$f_2 = 1$$
$$f_n = f_{n-1} + f_{n-2}$$

Fibonacci numbers describe the growth of a rabbit population.

Reformulate that as

```
fold1 = 1;
fold2 = 1;
fnew = fold1 + fold2;
```

After that, discard `fold2`, which is no longer needed, and set `fold2` to `fold1` and `fold1` to `fnew`. Repeat an appropriate number of times.

Implement a program that prompts the user for an integer *n* and prints the *n*th Fibonacci number, using the above algorithm.

■■■ **P4.17** *Factoring of integers*. Write a program that asks the user for an integer and then prints out all its factors. For example, when the user enters 150, the program should print

```
2
3
5
5
```

■■■ **P4.18** *Prime numbers*. Write a program that prompts the user for an integer and then prints out all prime numbers up to that integer. For example, when the user enters 20, the program should print

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

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

■ **P4.19** Write a program that prints a multiplication table, like this:

```
  1   2   3   4   5   6   7   8   9  10
  2   4   6   8  10  12  14  16  18  20
  3   6   9  12  15  18  21  24  27  30
  . . .
 10  20  30  40  50  60  70  80  90 100
```

■■ **P4.20** Write a program that reads an integer and displays, using asterisks, a filled and hollow square, placed next to each other. For example if the side length is 5, the program should display

```
***** *****
***** *   *
***** *   *
***** *   *
***** *****
```

■■ **P4.21** Write a program that reads an integer and displays, using asterisks, a filled diamond of the given side length. For example, if the side length is 4, the program should display

```
   *
  ***
 *****
*******
 *****
  ***
   *
```

■■■ **P4.22** *The game of Nim*. This is a well-known game with a number of variants. The following variant has an interesting winning strategy. Two players alternately take marbles from a pile. In each move, a player chooses how many marbles to take. The player must take at least one but at most half of the marbles. Then the other player takes a turn. The player who takes the last marble loses.

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

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

■■ **P4.23** *The Drunkard's Walk.* A drunkard in a grid of streets randomly picks one of four directions and stumbles to the next intersection, then again randomly picks one of four directions, and so on. You might think that on average the drunkard doesn't move very far because the choices cancel each other out, but that is actually not the case.

Represent locations as integer pairs (x, y). Implement the drunkard's walk over 100 intersections, starting at (0, 0), and print the ending location.

■■ **P4.24** *The Monty Hall Paradox*. Marilyn vos Savant described the following problem (loosely based on a game show hosted by Monty Hall) in a popular magazine: "Suppose you're on a game show, and you're given the choice of three doors: Behind one door is a car; behind the others, goats. You pick a door, say No. 1, and the host, who knows what's behind the doors, opens another door, say No. 3, which has a goat. He then says to you, "Do you want to pick door No. 2?" Is it to your advantage to switch your choice?"

Ms. vos Savant proved that it is to your advantage, but many of her readers, including some mathematics professors, disagreed, arguing that the probability would not change because another door was opened.

Your task is to simulate this game show. In each iteration, randomly pick a door number between 1 and 3 for placing the car. Randomly have the player pick a door. Randomly have the game show host pick a door having a goat (but not the door that the player picked). Increment a counter for strategy 1 if the player wins by switching to the host's choice, and increment a counter for strategy 2 if the player wins by sticking with the original choice. Run 1,000 iterations and print both counters.

■ **P4.25** A simple random generator is obtained by the formula

$$r_{\text{new}} = (a \cdot r_{\text{old}} + b)\%m$$

and then setting r_{old} to r_{new}. If m is chosen as 2^{32}, then you can compute

$$r_{\text{new}} = a \cdot r_{\text{old}} + b$$

because the truncation of an overflowing result to the `int` type is equivalent to computing the remainder.

Write a program that asks the user to enter a seed value for r_{old}. (Such a value is often called a *seed*). Then print the first 100 random integers generated by this formula, using $a = 32310901$ and $b = 1729$.

■■ **P4.26** *The Buffon Needle Experiment*. The following experiment was devised by Comte Georges-Louis Leclerc de Buffon (1707–1788), a French naturalist. A needle of length 1 inch is dropped onto paper that is ruled with lines 2 inches apart. If the needle drops onto a line, we count it as a *hit*. (See Figure 9.) Buffon discovered that the quotient *tries*/*hits* approximates π.

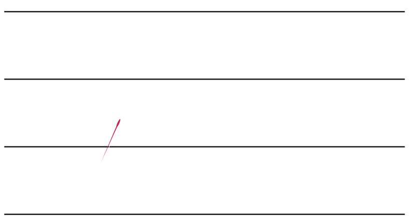


Figure 9
The Buffon Needle Experiment

For the Buffon needle experiment, 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.

Generate the *lower* point of the needle. Its x-coordinate is irrelevant, and you may assume its y-coordinate y_{low} to be any random number between 0 and 2. The angle α between the needle and the x-axis can be any value between 0 degrees and 180 degrees (π radians). The upper end of the needle has y-coordinate

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

The needle is a hit if y_{high} is at least 2, as shown in Figure 10. Stop after 10,000 tries and print the quotient *tries*/*hits*. (This program is not suitable for computing the value of π. You need π in the computation of the angle.)

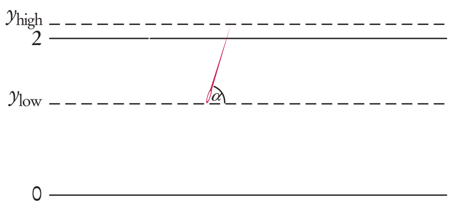


Figure 10
A Hit in the Buffon Needle Experiment

■■ Business P4.27 *Currency conversion*. Write a program that first asks the user to type today's price for one dollar in Japanese yen, then reads U.S. dollar values and converts each to yen. Use 0 as a sentinel.

■■ Business P4.28 Write a program that first asks the user to type in today's price of one dollar in Japanese yen, then reads U.S. dollar values and converts each to Japanese yen. Use 0 as the sentinel value to denote the end of dollar inputs. Then the program reads a sequence of yen amounts and converts them to dollars. The second sequence is terminated by another zero value.

■■ Business P4.29 Your company has shares of stock it would like to sell when their value exceeds a certain target price. Write a program that reads the target price and then reads the current stock price until it is at least the target price. Your program should use a `Scanner` to read a sequence of `double` values from standard input. Once the minimum is reached, the program should report that the stock price exceeds the target price.

■■ Business P4.30 Write an application to pre-sell a limited number of cinema tickets. Each buyer can buy as many as 4 tickets. No more than 100 tickets can be sold. Implement a program called `TicketSeller` that prompts the user for the desired number of tickets and then displays the number of remaining tickets. Repeat until all tickets have been sold, and then display the total number of buyers.

■■ Business P4.31 You need to control the number of people who can be in an oyster bar at the same time. Groups of people can always leave the bar, but a group cannot enter the bar if they would make the number of people in the bar exceed the maximum of 100 occupants. Write a program that reads the sizes of the groups that arrive or depart. Use negative numbers for departures. After each input, display the current number of occupants. As soon as the bar holds the maximum number of people, report that the bar is full and exit the program.

■■■ Business P4.32 *Credit Card Number Check*. The last digit of a credit card number is the *check digit*, which protects against transcription errors such as an error in a single digit or switching two digits. The following method is used to verify actual credit card numbers but, for simplicity, we will describe it for numbers with 8 digits instead of 16:

- Starting from the rightmost digit, form the sum of every other digit. For example, if the credit card number is 4358 9795, then you form the sum 5 + 7 + 8 + 3 = 23.
- Double each of the digits that were not included in the preceding step. Add all digits of the resulting numbers. For example, with the number given above, doubling the digits, starting with the next-to-last one, yields 18 18 10 8. Adding all digits in these values yields 1 + 8 + 1 + 8 + 1 + 0 + 8 = 27.
- Add the sums of the two preceding steps. If the last digit of the result is 0, the number is valid. In our case, 23 + 27 = 50, so the number is valid.

Write a program that implements this algorithm. The user should supply an 8-digit number, and you should print out whether the number is valid or not. If it is not valid, you should print the value of the check digit that would make it valid.

■■ Science P4.33 In a predator-prey simulation, you compute the populations of predators and prey, using the following equations:

$$prey_{n+1} = prey_n \times \left(1 + A - B \times pred_n\right)$$
$$pred_{n+1} = pred_n \times \left(1 - C + D \times prey_n\right)$$

Here, A is the rate at which prey birth exceeds natural death, B is the rate of predation, C is the rate at which predator deaths exceed births without food, and D represents predator increase in the presence of food.

Write a program that prompts users for these rates, the initial population sizes, and the number of periods. Then print the populations for the given number of periods. As inputs, try $A = 0.1$, $B = C = 0.01$, and $D = 0.00002$ with initial prey and predator populations of 1,000 and 20.

■■ Science P4.34 *Projectile flight*. Suppose a cannonball is propelled straight into the air with a starting velocity v_0. Any calculus book will state that the position of the ball after t seconds is $s(t) = -\frac{1}{2}gt^2 + v_0 t$, where $g = 9.81 \text{ m/s}^2$ is the gravitational force of the earth. No calculus textbook ever mentions why someone would want to carry out such an obviously dangerous experiment, so we will do it in the safety of the computer.

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

```
const double DELTA_T = 0.01;
```

and update the position by

```
s = s + v * DELTA_T;
```

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

```
v = v - g * DELTA_T;
```

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

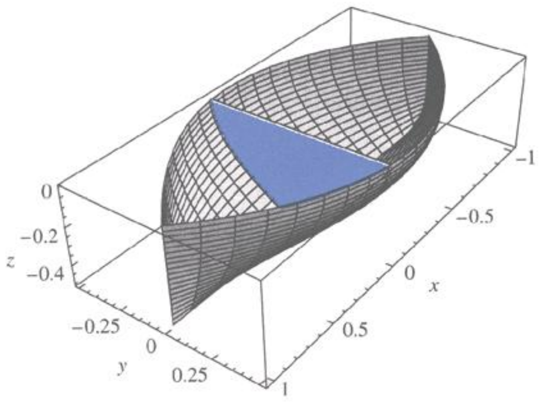
Now run the simulation until the cannonball falls back to the earth. Get the initial velocity as an input (100 m/s is a good value). Update the position and velocity 100 times per second, but print out the position only every full second. Also printout the values from the exact formula $s(t) = -\frac{1}{2}gt^2 + v_0 t$ for comparison.

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

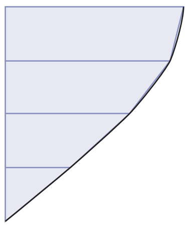
■■■ Science P4.35 A simple model for the hull of a ship is given by

$$|y| = \frac{B}{2}\left[1 - \left(\frac{2x}{L}\right)^2\right]\left[1 - \left(\frac{z}{T}\right)^2\right]$$

where B is the beam, L is the length, and T is the draft. (*Note:* There are two values of y for each x and z because the hull is symmetric from starboard to port.)

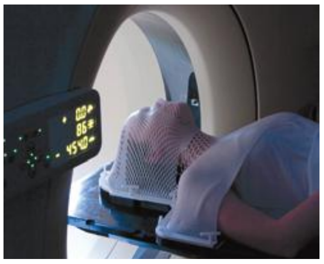


The cross-sectional area at a point x is called the "section" in nautical parlance. To compute it, let z go from 0 to $-T$ in n increments, each of size T/n. For each value of z, compute the value for y. Then sum the areas of trapezoidal strips. At right are the strips where $n = 4$.



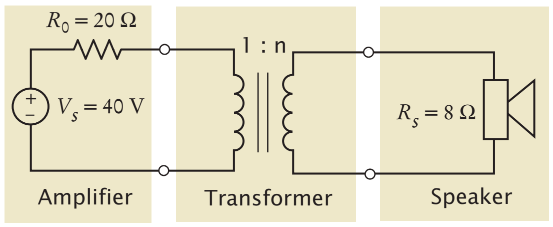
Write a program that reads in values for B, L, T, x, and n and then prints out the cross-sectional area at x.

■ Science P4.36 Radioactive decay of radioactive materials can be modeled by the equation $A = A_0 e^{-t(\log 2/h)}$, where A is the amount of the material at time t, A_0 is the amount at time 0, and h is the half-life.



Technetium-99 is a radioisotope that is used in imaging of the brain. It has a half-life of 6 hours. Your program should display the relative amount A / A_0 in a patient body every hour for 24 hours after receiving a dose.

■■■ Science P4.37 The photo at left shows an electric device called a "transformer". Transformers are often constructed by wrapping coils of wire around a ferrite core. The figure below illustrates a situation that occurs in various audio devices such as cell phones and music players. In this circuit, a transformer is used to connect a speaker to the output of an audio amplifier.



The symbol used to represent the transformer is intended to suggest two coils of wire. The parameter n of the transformer is called the “turns ratio” of the transformer. (The number of times that a wire is wrapped around the core to form a coil is called the number of turns in the coil. The turns ratio is literally the ratio of the number of turns in the two coils of wire.)

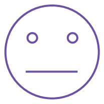
When designing the circuit, we are concerned primarily with the value of the power delivered to the speakers—that power causes the speakers to produce the sounds we want to hear. Suppose we were to connect the speakers directly to the amplifier without using the transformer. Some fraction of the power available from the amplifier would get to the speakers. The rest of the available power would be lost in the amplifier itself. The transformer is added to the circuit to increase the fraction of the amplifier power that is delivered to the speakers.

The power, P_s, delivered to the speakers is calculated using the formula

$$P_s = R_s \left(\frac{nV_s}{n^2 R_0 + R_s} \right)^2$$

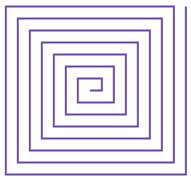
Write a program that models the circuit shown and varies the turns ratio from 0.01 to 2 in 0.01 increments, then determines the value of the turns ratio that maximizes the power delivered to the speakers.

■ Graphics P4.38 Write a program to plot the following face.



■ Graphics P4.39 Write a graphical application that displays a checkerboard with 64 squares, alternating white and black.

■■■ Graphics P4.40 Write a graphical application that draws a spiral, such as the following:



■■ Graphics P4.41 It is easy and fun to draw graphs of curves with the Java graphics library. Simply draw 100 line segments joining the points $(x, f(x))$ and $(x + d, f(x + d))$, where x ranges from $x_{\min}$ to $x_{\max}$ and $d = (x_{\max} - x_{\min})/100$.

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

■■■ Graphics P4.42 Draw a picture of the “four-leaved rose” whose equation in polar coordinates is $r = \cos(2\theta)$. Let θ go from 0 to 2π in 100 steps. Each time, compute r and then compute the (x, y) coordinates from the polar coordinates by using the formula

$$x = r \cdot \cos(\theta), y = r \cdot \sin(\theta)$$

ANSWERS TO SELF-CHECK QUESTIONS

1. 23 years.

2. 7 years.

3. Add a statement

```
System.out.println(balance);
```

as the last statement in the `while` loop.

4. The program prints the same output. This is because the balance after 14 years is slightly below $20,000, and after 15 years, it is slightly above $20,000.

5.
```
2 4 8 16 32 64 128
```

Note that the value 128 is printed even though it is larger than 100.

6.

n	output
5	
~~4~~	4
~~3~~	3
~~2~~	2
~~1~~	1
~~0~~	0
-1	-1

7.

n	output
~~1~~	1,
~~2~~	1, 2,
~~3~~	1, 2, 3,
4	

There is a comma after the last value. Usually, commas are between values only.

8.

a	n	r	i
2	4	~~1~~	~~1~~
		~~2~~	~~2~~
		~~4~~	~~3~~
		~~8~~	~~4~~
		16	5

The code computes a^n.

9.

n	output
~~1~~	1
~~11~~	11
~~21~~	21
~~31~~	31
~~41~~	41
~~51~~	51
~~61~~	61
...	

This is an infinite loop. `n` is never equal to 50.

10.

count	temp
1	123
2	12.3
3	1.23

This yields the correct answer. The number 123 has 3 digits.

count	temp
1	100
2	10.0

This yields the wrong answer. The number 100 also has 3 digits. The loop condition should have been

```
while (temp >= 10)
```

11.
```
int year = 1;
while (year <= nyears)
{
   double interest = balance * RATE / 100;
   balance = balance + interest;
   System.out.printf("%4d %10.2f\n",
      year, balance);
   year++;
}
```

12. 11 numbers: 10 9 8 7 6 5 4 3 2 1 0

13.
```
for (int i = 10; i <= 20; i = i + 2)
{
   System.out.println(i);
}
```

14.
```
int sum = 0;
for (int i = 1; i <= n; i++)
{
   sum = sum + i;
}
```

15.
```
for (int year = 1;
  balance <= 2 * INITIAL_BALANCE; year++)
```

However, it is best not to use a `for` loop in this case because the loop condition does not relate to the `year` variable. A `while` loop would be a better choice.

16.
```
do
{
   System.out.print(
      "Enter a value between 0 and 100: ");
   value = in.nextInt();
}
while (value < 0 || value > 100);
```

17.
```
int value = 100;
while (value >= 100)
{
   System.out.print("Enter a value < 100: ");
   value = in.nextInt();
}
```