Program Run

```
        Country    Gold  Silver  Bronze   Total
         Canada       1       0       1       2
          China       1       1       0       2
        Germany       0       0       1       1
          Korea       1       0       0       1
          Japan       0       1       1       2
         Russia       0       1       1       2
  United States       1       1       0       2
```

SELF CHECK

30. What results do you get if you total the columns in our sample data?

31. Consider an 8 × 8 array for a board game:

```
int[][] board = new int[8][8];
```

Using two nested loops, initialize the board so that zeroes and ones alternate, as on a checkerboard:

```
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
. . .
1 0 1 0 1 0 1 0
```

Hint: Check whether `i + j` is even.

32. Declare a two-dimensional array for representing a tic-tac-toe board. The board has three rows and columns and contains strings `"x"`, `"o"`, and `" "`.

33. Write an assignment statement to place an `"x"` in the upper-right corner of the tic-tac-toe board in Self Check 32.

34. Which elements are on the diagonal joining the upper-left and the lower-right corners of the tic-tac-toe board in Self Check 32?

Practice It Now you can try these exercises at the end of the chapter: R7.28, E7.15, E7.16.

WORKED EXAMPLE 7.2 **A World Population Table**

Learn how to print world population data in a table with row and column headers, and with totals for each of the data columns. Go to `wiley.com/go/javaexamples` and download the file for Worked Example 7.2.

Special Topic 7.3

Two-Dimensional Arrays with Variable Row Lengths

When you declare a two-dimensional array with the command

```
int[][] a = new int[3][3];
```

you get a 3 × 3 matrix that can store 9 elements:

```
a[0][0] a[0][1] a[0][2]
a[1][0] a[1][1] a[1][2]
a[2][0] a[2][1] a[2][2]
```

In this matrix, all rows have the same length.

In Java it is possible to declare arrays in which the row length varies. For example, you can store an array that has a triangular shape, such as:

```
b[0][0]
b[1][0] b[1][1]
b[2][0] b[2][1] b[2][2]
```

To allocate such an array, you must work harder. First, you allocate space to hold three rows. Indicate that you will manually set each row by leaving the second array index empty:

```
double[][] b = new double[3][];
```

Then allocate each row separately (see Figure 16):

```
for (int i = 0; i < b.length; i++)
{
   b[i] = new double[i + 1];
}
```

You can access each array element as `b[i][j]`. The expression `b[i]` selects the `i`th row, and the `[j]` operator selects the `j`th element in that row.

Note that the number of rows is `b.length`, and the length of the `i`th row is `b[i].length`. For example, the following pair of loops prints a ragged array:

```
for (int i = 0; i < b.length; i++)
{
   for (int j = 0; j < b[i].length; j++)
   {
      System.out.print(b[i][j]);
   }
   System.out.println();
}
```

Alternatively, you can use two enhanced `for` loops:

```
for (double[] row : b)
{
   for (double element : row)
   {
      System.out.print(element);
   }
   System.out.println();
}
```

Naturally, such "ragged" arrays are not very common.

Java implements plain two-dimensional arrays in exactly the same way as ragged arrays: as arrays of one-dimensional arrays. The expression `new int[3][3]` automatically allocates an array of three rows, and three arrays for the rows' contents.

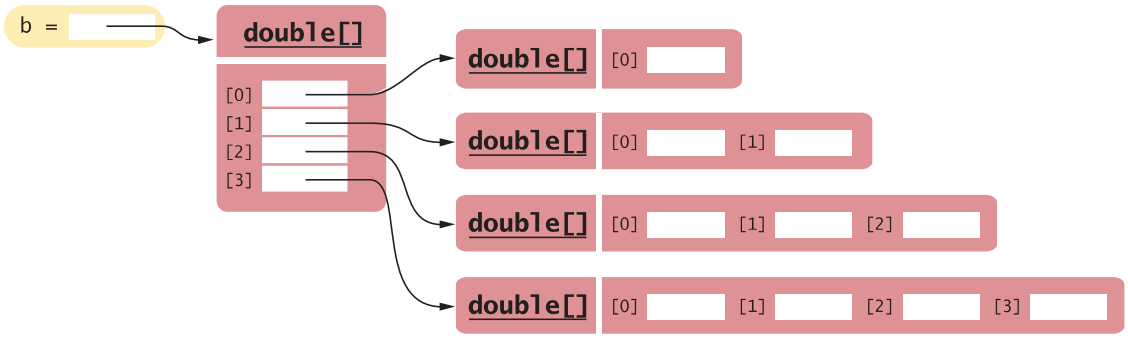


Figure 16 A Triangular Array

Special Topic 7.4

Multidimensional Arrays

You can declare arrays with more than two dimensions. For example, here is a three-dimensional array:

```
int[][][] rubiksCube = new int[3][3][3];
```

Each array element is specified by three index values:

```
rubiksCube[i][j][k]
```

7.7 Array Lists

An array list stores a sequence of values whose size can change.

When you write a program that collects inputs, you don't always know how many inputs you will have. In such a situation, an **array list** offers two significant advantages:

- Array lists can grow and shrink as needed.
- The `ArrayList` class supplies methods for common tasks, such as inserting and removing elements.

In the following sections, you will learn how to work with array lists.

An array list expands to hold as many elements as needed.

Syntax 7.4 Array Lists

Syntax

To construct an array list:
```
new ArrayList<typeName>()
```

To access an element:
```
arraylistReference.get(index)
arraylistReference.set(index, value)
```

Variable type — Variable name — An array list object of size 0

```
ArrayList<String> friends = new ArrayList<String>();
```

```
friends.add("Cindy");
String name = friends.get(i);
friends.set(i, "Harry");
```

The `add` method appends an element to the array list, increasing its size.

Use the `get` and `set` methods to access an element.

The index must be ≥ 0 and < `friends.size()`.

7.7.1 Declaring and Using Array Lists

The following statement declares an array list of strings:

```
ArrayList<String> names = new ArrayList<String>();
```

The ArrayList class is a generic class: ArrayList<*Type*> collects elements of the specified type.

The `ArrayList` class is contained in the `java.util` package. In order to use array lists in your program, you need to use the statement `import java.util.ArrayList`.

The type `ArrayList<String>` denotes an array list of `String` elements. The angle brackets around the `String` type tell you that `String` is a **type parameter**. You can replace `String` with any other class and get a different array list type. For that reason, `ArrayList` is called a **generic class**. However, you cannot use primitive types as type parameters—there is no `ArrayList<int>` or `ArrayList<double>`. Section 7.7.4 shows how you can collect numbers in an array list.

It is a common error to forget the initialization:

```
ArrayList<String> names;
names.add("Harry"); // Error—names not initialized
```

Here is the proper initialization:

```
ArrayList<String> names = new ArrayList<String>();
```

Note the `()` after `new ArrayList<String>` on the right-hand side of the initialization. It indicates that the **constructor** of the `ArrayList<String>` class is being called.

When the `ArrayList<String>` is first constructed, it has size 0. You use the `add` method to add an element to the end of the array list.

```
names.add("Emily"); // Now names has size 1 and element "Emily"
names.add("Bob"); // Now names has size 2 and elements "Emily", "Bob"
names.add("Cindy"); // names has size 3 and elements "Emily", "Bob", and "Cindy"
```

Use the size method to obtain the current size of an array list.

The size increases after each call to `add` (see Figure 17). The `size` method yields the current size of the array list.

To obtain an array list element, use the `get` method, not the `[]` operator. As with arrays, index values start at 0. For example, `names.get(2)` retrieves the name with index 2, the third element in the array list:

Use the get and set methods to access an array list element at a given index.

```
String name = names.get(2);
```

As with arrays, it is an error to access a nonexistent element. A very common bounds error is to use the following:

```
int i = names.size();
name = names.get(i);  // Error
```

The last valid index is `names.size() - 1`.

To set an array list element to a new value, use the `set` method:

```
names.set(2, "Carolyn");
```

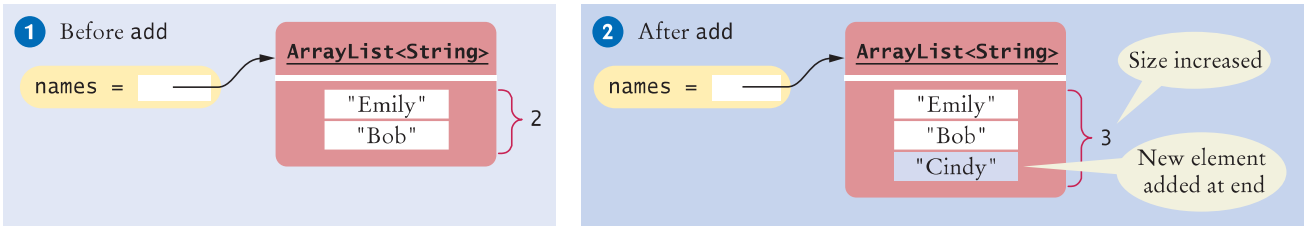


Figure 17 Adding an Array List Element with `add`

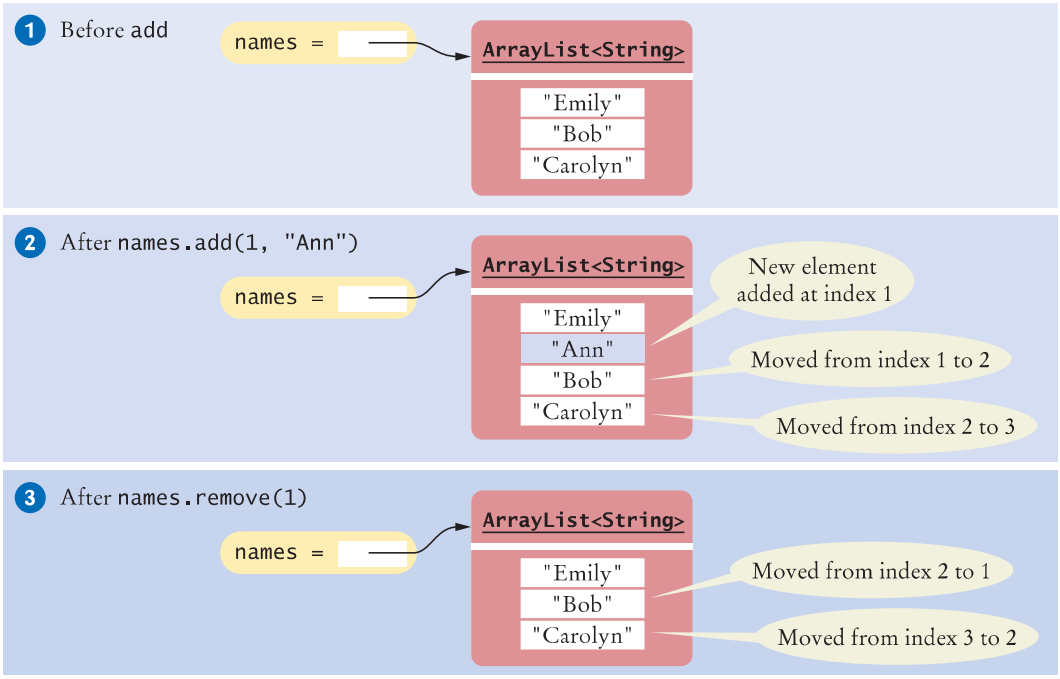


Figure 18 Adding and Removing Elements in the Middle of an Array List

An array list has methods for adding and removing elements in the middle.

Use the add and remove methods to add and remove array list elements.

This call sets position 2 of the `names` array list to `"Carolyn"`, overwriting whatever value was there before.

The `set` method overwrites existing values. It is different from the `add` method, which adds a new element to the array list.

You can insert an element in the middle of an array list. For example, the call `names.add(1, "Ann")` adds a new element at position 1 and moves all elements with index 1 or larger by one position. After each call to the `add` method, the size of the array list increases by 1 (see Figure 18).

Conversely, the `remove` method removes the element at a given position, moves all elements after the removed element down by one position, and reduces the size of the array list by 1. Part 3 of Figure 18 illustrates the result of `names.remove(1)`.

With an array list, it is very easy to get a quick printout. Simply pass the array list to the `println` method:

```
System.out.println(names); // Prints [Emily, Bob, Carolyn]
```

7.7.2 Using the Enhanced for Loop with Array Lists

You can use the enhanced `for` loop to visit all elements of an array list. For example, the following loop prints all names:

```
ArrayList<String> names = . . . ;
for (String name : names)
{
   System.out.println(name);
}
```

This loop is equivalent to the following basic `for` loop:

```
for (int i = 0; i < names.size(); i++)
{
```

```
      String name = names.get(i);
      System.out.println(name);
   }
```

7.7.3 Copying Array Lists

As with arrays, you need to remember that array list variables hold references. Copying the reference yields two references to the same array list (see Figure 19).

```
ArrayList<String> friends = names;
friends.add("Harry");
```

Now both `names` and `friends` reference the same array list to which the string `"Harry"` was added.

If you want to make a copy of an array list, construct the copy and pass the original list into the constructor:

```
ArrayList<String> newNames = new ArrayList<String>(names);
```

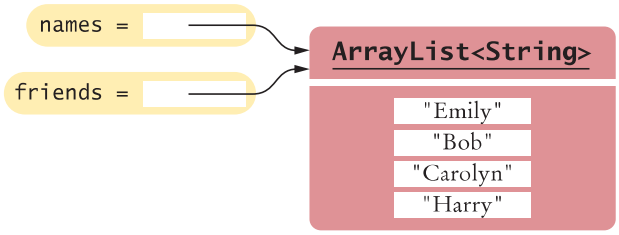


Figure 19
Copying an Array List Reference

Table 2 Working with Array Lists

`ArrayList<String> names = new ArrayList<String>();`	Constructs an empty array list that can hold strings.
`names.add("Ann");` `names.add("Cindy");`	Adds elements to the end.
`System.out.println(names);`	Prints `[Ann, Cindy]`.
`names.add(1, "Bob");`	Inserts an element at index 1. `names` is now `[Ann, Bob, Cindy]`.
`names.remove(0);`	Removes the element at index 0. `names` is now `[Bob, Cindy]`.
`names.set(0, "Bill");`	Replaces an element with a different value. `names` is now `[Bill, Cindy]`.
`String name = names.get(i);`	Gets an element.
`String last = names.get(names.size() - 1);`	Gets the last element.

`ArrayList<Integer> squares = new ArrayList<Integer>();`
`for (int i = 0; i < 10; i++)`
`{`
`  squares.add(i * i);`
`}`
	Constructs an array list holding the first ten squares.

7.7.4 Wrappers and Auto-boxing

Like truffles that must be in a wrapper to be sold, a number must be placed in a wrapper to be stored in an array list.

In Java, you cannot directly insert primitive type values—numbers, characters, or `boolean` values—into array lists. For example, you cannot form an `ArrayList<double>`. Instead, you must use one of the **wrapper classes** shown in the following table.

Primitive Type	Wrapper Class
`byte`	`Byte`
`boolean`	`Boolean`
`char`	`Character`
`double`	`Double`
`float`	`Float`
`int`	`Integer`
`long`	`Long`
`short`	`Short`

To collect numbers in array lists, you must use wrapper classes.

For example, to collect `double` values in an array list, you use an `ArrayList<Double>`. Note that the wrapper class names start with uppercase letters, and that two of them differ from the names of the corresponding primitive type: `Integer` and `Character`.

Conversion between primitive types and the corresponding wrapper classes is automatic. This process is called **auto-boxing** (even though *auto-wrapping* would have been more consistent).

For example, if you assign a `double` value to a `Double` variable, the number is automatically "put into a box" (see Figure 20).

```
Double wrapper = 29.95;
```

Conversely, wrapper values are automatically "unboxed" to primitive types:

```
double x = wrapper;
```

Because boxing and unboxing is automatic, you don't need to think about it. Simply remember to use the wrapper type when you declare array lists of numbers. From then on, use the primitive type and rely on auto-boxing.

```
ArrayList<Double> values = new ArrayList<Double>();
values.add(29.95);
double x = values.get(0);
```

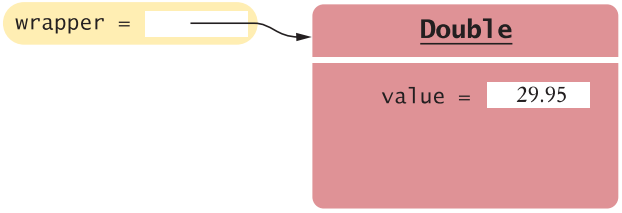


Figure 20 A Wrapper Class Variable

7.7.5 Using Array Algorithms with Array Lists

The array algorithms in Section 7.3 can be converted to array lists simply by using the array list methods instead of the array syntax (see Table 3 on page 354). For example, this code snippet finds the largest element in an array:

```
double largest = values[0];
for (int i = 1; i < values.length; i++)
{
   if (values[i] > largest)
   {
      largest = values[i];
   }
}
```

Here is the same algorithm, now using an array list:

```
double largest = values.get(0);
for (int i = 1; i < values.size(); i++)
{
   if (values.get(i) > largest)
   {
      largest = values.get(i);
   }
}
```

7.7.6 Storing Input Values in an Array List

When you collect an unknown number of inputs, array lists are *much* easier to use than arrays. Simply read inputs and add them to an array list:

```
ArrayList<Double> inputs = new ArrayList<Double>();
while (in.hasNextDouble())
{
   inputs.add(in.nextDouble());
}
```

7.7.7 Removing Matches

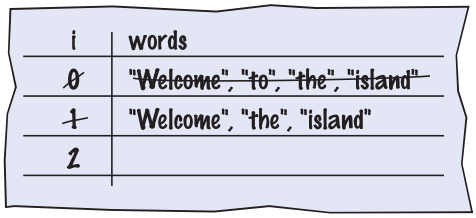
It is easy to remove elements from an array list, by calling the `remove` method. A common processing task is to remove all elements that match a particular condition. Suppose, for example, that we want to remove all strings of length < 4 from an array list.

Of course, you traverse the array list and look for matching elements:

```
ArrayList<String> words = ...;
for (int i = 0; i < words.size(); i++)
{
   String word = words.get(i);
   if (word.length() < 4)
   {
      Remove the element at index i.
   }
}
```

But there is a subtle problem. After you remove the element, the `for` loop increments `i`, skipping past the *next* element.

Consider this concrete example, where `words` contains the strings `"Welcome"`, `"to"`, `"the"`, `"island!"`. When `i` is 1, we remove the word `"to"` at index 1. Then `i` is incremented to 2, and the word `"the"`, which is now at position 1, is never examined.



We should not increment the index when removing a word. The appropriate pseudocode is

```
If the element at index i matches the condition
   Remove the element.
Else
   Increment i.
```

Because we don't always increment the index, a `for` loop is not appropriate for this algorithm. Instead, use a `while` loop:

```
int i = 0;
while (i < words.size())
{
   String word = words.get(i);
   if (word.length() < 4)
   {
      words.remove(i);
   }
   else
   {
      i++;
   }
}
```

7.7.8 Choosing Between Array Lists and Arrays

For most programming tasks, array lists are easier to use than arrays. Array lists can grow and shrink. On the other hand, arrays have a nicer syntax for element access and initialization.

Which of the two should you choose? Here are some recommendations.

- If the size of a collection never changes, use an array.
- If you collect a long sequence of primitive type values and you are concerned about efficiency, use an array.
- Otherwise, use an array list.

FULL CODE EXAMPLE

Go to `wiley.com/go/javacode` to download a version of the Student class using an array list.

The following program shows how to mark the largest value in a sequence of values. This program uses an array list. Note how the program is an improvement over the array version on page 329. This program can process input sequences of arbitrary length.

Table 3 Comparing Array and Array List Operations

Operation	Arrays	Array Lists
Get an element.	`x = values[4];`	`x = values.get(4)`
Replace an element.	`values[4] = 35;`	`values.set(4, 35);`
Number of elements.	`values.length`	`values.size()`
Number of filled elements.	`currentSize` (companion variable, see Section 7.1.4)	`values.size()`
Remove an element.	See Section 7.3.6	`values.remove(4);`
Add an element, growing the collection.	See Section 7.3.7	`values.add(35);`
Initializing a collection.	`int[] values = { 1, 4, 9 };`	No initializer list syntax; call `add` three times.

section_7/LargestInArrayList.java

```
1  import java.util.ArrayList;
2  import java.util.Scanner;
3
4  /**
5     This program reads a sequence of values and prints them, marking the largest value.
6  */
7  public class LargestInArrayList
8  {
9     public static void main(String[] args)
10    {
11       ArrayList<Double> values = new ArrayList<Double>();
12
13       // Read inputs
14
15       System.out.println("Please enter values, Q to quit:");
16       Scanner in = new Scanner(System.in);
17       while (in.hasNextDouble())
18       {
19          values.add(in.nextDouble());
20       }
21
22       // Find the largest value
23
24       double largest = values.get(0);
25       for (int i = 1; i < values.size(); i++)
26       {
27          if (values.get(i) > largest)
28          {
29             largest = values.get(i);
30          }
31       }
32
33       // Print all values, marking the largest
34
```

```
35         for (double element : values)
36         {
37            System.out.print(element);
38            if (element == largest)
39            {
40               System.out.print(" <== largest value");
41            }
42            System.out.println();
43         }
44      }
45   }
```

Program Run

```
Please enter values, Q to quit:
35 80 115 44.5 Q
35
80
115 <== largest value
44.5
```

35. Declare an array list `primes` of integers that contains the first five prime numbers (2, 3, 5, 7, and 11).

36. Given the array list `primes` declared in Self Check 35, write a loop to print its elements in reverse order, starting with the last element.

37. What does the array list `names` contain after the following statements?

```
ArrayList<String> names = new ArrayList<String>;
names.add("Bob");
names.add(0, "Ann");
names.remove(1);
names.add("Cal");
```

38. What is wrong with this code snippet?

```
ArrayList<String> names;
names.add(Bob);
```

39. Consider this method that appends the elements of one array list to another:

```
public void append(ArrayList<String> target, ArrayList<String> source)
{
   for (int i = 0; i < source.size(); i++)
   {
      target.add(source.get(i));
   }
}
```

What are the contents of `names1` and `names2` after these statements?

```
ArrayList<String> names1 = new ArrayList<String>();
names1.add("Emily");
names1.add("Bob");
names1.add("Cindy");
ArrayList<String> names2 = new ArrayList<String>();
names2.add("Dave");
append(names1, names2);
```

40. Suppose you want to store the names of the weekdays. Should you use an array list or an array of seven strings?

41. The ch07/section_7 directory of your source code contains an alternate implementation of the problem solution in How To 7.1 on page 334. Compare the array and array list implementations. What is the primary advantage of the latter?

Practice It Now you can try these exercises at the end of the chapter: R7.10, R7.32, E7.17, E7.19.

Common Error 7.4

Length and Size

Unfortunately, the Java syntax for determining the number of elements in an array, an array list, and a string is not at all consistent. It is a common error to confuse these. You just have to remember the correct syntax for every data type.

Data Type	Number of Elements
Array	`a.length`
Array list	`a.size()`
String	`a.length()`

Special Topic 7.5

The Diamond Syntax in Java 7

Java 7 introduces a convenient syntax enhancement for declaring array lists and other generic classes. In a statement that declares and constructs an array list, you need not repeat the type parameter in the constructor. That is, you can write

```
ArrayList<String> names = new ArrayList<>();
```

instead of

```
ArrayList<String> names = new ArrayList<String>();
```

This shortcut is called the "diamond syntax" because the empty brackets <> look like a diamond shape.

7.8 Regression Testing

A test suite is a set of tests for repeated testing.

It is a common and useful practice to make a new test whenever you find a program bug. You can use that test to verify that your bug fix really works. Don't throw the test away; feed it to the next version after that and all subsequent versions. Such a collection of test cases is called a **test suite**.

You will be surprised how often a bug that you fixed will reappear in a future version. This is a phenomenon known as *cycling*. Sometimes you don't quite understand the reason for a bug and apply a quick fix that appears to work. Later, you apply a different quick fix that solves a second problem but makes the first problem appear again. Of course, it is always best to think through what really causes a bug and fix the root cause instead of doing a sequence of "Band-Aid" solutions. If you don't succeed in doing that, however, you at least want to have an honest appraisal of how well the program works. By keeping all old test cases around and testing them against every

Regression testing involves repeating previously run tests to ensure that known failures of prior versions do not appear in new versions of the software.

new version, you get that feedback. The process of checking each version of a program against a test suite is called **regression testing**.

How do you organize a suite of tests? An easy technique is to produce multiple tester classes, such as `ScoreTester1`, `ScoreTester2`, and so on, where each program runs with a separate set of test data. For example, here is a tester for the `Student` class:

```
public class ScoreTester1
{
   public static void main(String[] args)
   {
      Scanner in = new Scanner(System.in);
      Student fred = new Student(100);
      fred.addScore(10);
      fred.addScore(20);
      fred.addScore(5);
      System.out.println("Final score: " + fred.finalScore());
      System.out.println("Expected: 30");
   }
}
```

Another useful approach is to provide a generic tester, and feed it inputs from multiple files, as in the following.

section_8/ScoreTester.java

```
1  import java.util.Scanner;
2
3  public class ScoreTester
4  {
5     public static void main(String[] args)
6     {
7        Scanner in = new Scanner(System.in);
8        double expected = in.nextDouble();
9        Student fred = new Student(100);
10       while (in.hasNextDouble())
11       {
12          if (!fred.addScore(in.nextDouble()))
13          {
14             System.out.println("Too many scores.");
15             return;
16          }
17       }
18       System.out.println("Final score: " + fred.finalScore());
19       System.out.println("Expected: " + expected);
20    }
21 }
```

The program reads the expected result and the scores. By running the program with different inputs, we can test different scenarios.

Of course, it would be tedious to type in the input values by hand every time the test is executed. It is much better to save the inputs in a file, such as the following:

section_8/input1.txt

```
30
10
20
5
```

When running the program from a shell window, one can link the input file to the input of a program, as if all the characters in the file had actually been typed by a user. Type the following command into a shell window:

```
java ScoreTester < input1.txt
```

The program is executed, but it no longer reads input from the keyboard. Instead, the `System.in` object (and the `Scanner` that reads from `System.in`) gets the input from the file `input1.txt`. We discussed this process, called input **redirection**, in Special Topic 6.2.

The output is still displayed in the console window:

Program Run

```
Final score: 30
Expected: 30
```

You can also redirect output. To capture the program's output in a file, use the command

```
java ScoreTester < input1.txt > output1.txt
```

This is useful for archiving test cases.

42. Suppose you modified the code for a method. Why do you want to repeat tests that already passed with the previous version of the code?
43. Suppose a customer of your program finds an error. What action should you take beyond fixing the error?
44. Why doesn't the `ScoreTester` program contain prompts for the inputs?

Practice It Now you can try these exercises at the end of the chapter: R7.34, R7.35.

Programming Tip 7.3

Batch Files and Shell Scripts

If you need to perform the same tasks repeatedly on the command line, then it is worth learning about the automation features offered by your operating system.

Under Windows, you use batch files to execute a number of commands automatically. For example, suppose you need to test a program by running three testers:

```
java ScoreTester1
java ScoreTester < input1.txt
java ScoreTester < input2.txt
```

Then you find a bug, fix it, and run the tests again. Now you need to type the three commands once more. There has to be a better way. Under Windows, put the commands in a text file and call it `test.bat`:

File test.bat

```
1  java ScoreTester1
2  java ScoreTester < input1.txt
3  java ScoreTester < input2.txt
```

Then you just type

```
test.bat
```

and the three commands in the batch file execute automatically.

Batch files are a feature of the operating system, not of Java. On Linux, Mac OS, and UNIX, shell scripts are used for the same purpose. In this simple example, you can execute the commands by typing

```
sh test.bat
```

There are many uses for batch files and shell scripts, and it is well worth it to learn more about their advanced features, such as parameters and loops.

Computing & Society 7.2 The Therac-25 Incidents

The Therac-25 is a computerized device to deliver radiation treatment to cancer patients (see the figure). Between June 1985 and January 1987, several of these machines delivered serious overdoses to at least six patients, killing some of them and seriously maiming the others.

The machines were controlled by a computer program. Bugs in the program were directly responsible for the overdoses. According to Leveson and Turner ("An Investigation of the Therac-25 Accidents," *IEEE Computer*, July 1993, pp. 18–41), the program was written by a single programmer, who had since left the manufacturing company producing the device and could not be located. None of the company employees interviewed could say anything about the educational level or qualifications of the programmer.

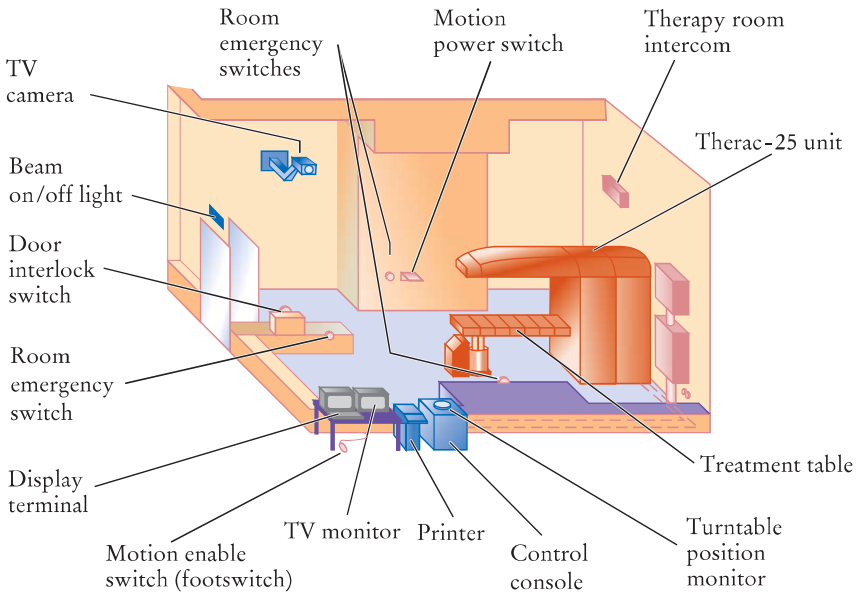
The investigation by the federal Food and Drug Administration (FDA) found that the program was poorly documented and that there was neither a specification document nor a formal test plan. (This should make you think. Do you have a formal test plan for your programs?)

The overdoses were caused by an amateurish design of the software that had to control different devices concurrently, namely the keyboard, the display, the printer, and of course the radiation device itself. Synchronization and data sharing between the tasks were done in an ad hoc way, even though safe multitasking techniques were known at the time. Had the programmer enjoyed a formal education that involved these techniques, or taken the effort to study the literature, a safer machine could have been built. Such a machine would have probably involved a commercial multitasking system, which might have required a more expensive computer.

The same flaws were present in the software controlling the predecessor model, the Therac-20, but that machine had hardware interlocks that mechanically prevented overdoses. The hardware safety devices were removed in the Therac-25 and replaced by checks in the software, presumably to save cost.

Frank Houston of the FDA wrote in 1985: "A significant amount of software for life-critical systems comes from small firms, especially in the medical device industry; firms that fit the profile of those resistant to or uninformed of the principles of either system safety or software engineering."

Who is to blame? The programmer? The manager who not only failed to ensure that the programmer was up to the task but also didn't insist on comprehensive testing? The hospitals that installed the device, or the FDA, for not reviewing the design process? Unfortunately, even today there are no firm standards of what constitutes a safe software design process.



Typical Therac-25 Facility

CHAPTER SUMMARY

Use arrays for collecting values.

- An array collects a sequence of values of the same type.
- Individual elements in an array are accessed by an integer index `i`, using the notation *array*`[i]`.
- An array element can be used like any variable.
- An array index must be at least zero and less than the size of the array.
- A bounds error, which occurs if you supply an invalid array index, can cause your program to terminate.
- Use the expression *array*`.length` to find the number of elements in an array.
- An array reference specifies the location of an array. Copying the reference yields a second reference to the same array.
- Arrays can occur as method arguments and return values.
- With a partially filled array, keep a companion variable for the current size.
- Avoid parallel arrays by changing them into arrays of objects.

Know when to use the enhanced `for` loop.

- You can use the enhanced `for` loop to visit all elements of an array.
- Use the enhanced `for` loop if you do not need the index values in the loop body.

Know and use common array algorithms.

- When separating elements, don't place a separator before the first element.
- A linear search inspects elements in sequence until a match is found.
- Before inserting an element, move elements to the end of the array *starting with the last one*.
- Use a temporary variable when swapping two elements.
- Use the `Arrays.copyOf` method to copy the elements of an array into a new array.

Combine and adapt algorithms for solving a programming problem.

- By combining fundamental algorithms, you can solve complex programming tasks.
- You should be familiar with the implementation of fundamental algorithms so that you can adapt them.

Discover algorithms by manipulating physical objects.

- Use a sequence of coins, playing cards, or toys to visualize an array of values.
- You can use paper clips as position markers or counters.

Use two-dimensional arrays for data that is arranged in rows and columns.

- Use a two-dimensional array to store tabular data.
- Individual elements in a two-dimensional array are accessed by using two index values, *array*`[i][j]`.

Use array lists for managing collections whose size can change.

- An array list stores a sequence of values whose size can change.
- The `ArrayList` class is a generic class: `ArrayList<`*Type*`>` collects elements of the specified type.
- Use the `size` method to obtain the current size of an array list.
- Use the `get` and `set` methods to access an array list element at a given index.
- Use the `add` and `remove` methods to add and remove array list elements.
- To collect numbers in array lists, you must use wrapper classes.

Describe the process of regression testing.

- A test suite is a set of tests for repeated testing.
- Regression testing involves repeating previously run tests to ensure that known failures of prior versions do not appear in new versions of the software.

STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

```
java.lang.Boolean
java.lang.Double
java.lang.Integer
java.util.Arrays
   copyOf
   toString
```

```
java.util.ArrayList<E>
   add
   get
   remove
   set
   size
```

REVIEW QUESTIONS

■■ **R7.1** Write code that fills an array `values` with each set of numbers below.

a.	1	2	3	4	5	6	7	8	9	10
b.	0	2	4	6	8	10	12	14	16	18	20
c.	1	4	9	16	25	36	49	64	81	100
d.	0	0	0	0	0	0	0	0	0	0
e.	1	4	9	16	9	7	4	9	11	
f.	0	1	0	1	0	1	0	1	0	1
g.	0	1	2	3	4	0	1	2	3	4

■■ **R7.2** Consider the following array:

```
int[] a = { 1, 2, 3, 4, 5, 4, 3, 2, 1, 0 };
```

What is the value of `total` after the following loops complete?

a.
```
int total = 0;
for (int i = 0; i < 10; i++) { total = total + a[i]; }
```

b.
```
int total = 0;
for (int i = 0; i < 10; i = i + 2) { total = total + a[i]; }
```

c.
```
int total = 0;
for (int i = 1; i < 10; i = i + 2) { total = total + a[i]; }
```

d.
```
int total = 0;
for (int i = 2; i <= 10; i++) { total = total + a[i]; }
```

e.
```
int total = 0;
for (int i = 1; i < 10; i = 2 * i) { total = total + a[i]; }
```

f.
```
int total = 0;
for (int i = 9; i >= 0; i--) { total = total + a[i]; }
```

g.
```
int total = 0;
for (int i = 9; i >= 0; i = i - 2) { total = total + a[i]; }
```

h.
```
int total = 0;
for (int i = 0; i < 10; i++) { total = a[i] - total; }
```

■■ **R7.3** Consider the following array:

```
int[] a = { 1, 2, 3, 4, 5, 4, 3, 2, 1, 0 };
```

What are the contents of the array `a` after the following loops complete?

a.
```
for (int i = 1; i < 10; i++) { a[i] = a[i - 1]; }
```

b.
```
for (int i = 9; i > 0; i--) { a[i] = a[i - 1]; }
```

c.
```
for (int i = 0; i < 9; i++) { a[i] = a[i + 1]; }
```

d.
```
for (int i = 8; i >= 0; i--) { a[i] = a[i + 1]; }
```

e.
```
for (int i = 1; i < 10; i++) { a[i] = a[i] + a[i - 1]; }
```

f.
```
for (int i = 1; i < 10; i = i + 2) { a[i] = 0; }
```

g.
```
for (int i = 0; i < 5; i++) { a[i + 5] = a[i]; }
```

h.
```
for (int i = 1; i < 5; i++) { a[i] = a[9 - i]; }
```

■■■ **R7.4** Write a loop that fills an array `values` with ten random numbers between 1 and 100. Write code for two nested loops that fill `values` with ten *different* random numbers between 1 and 100.

■■ **R7.5** Write Java code for a loop that simultaneously computes both the maximum and minimum of an array.

■ **R7.6** What is wrong with each of the following code segments?

a.
```
int[] values = new int[10];
for (int i = 1; i <= 10; i++)
{
   values[i] = i * i;
}
```

b.
```
int[] values;
for (int i = 0; i < values.length; i++)
{
   values[i] = i * i;
}
```

■■ **R7.7** Write enhanced `for` loops for the following tasks.

a. Printing all elements of an array in a single row, separated by spaces.

b. Computing the maximum of all elements in an array.

c. Counting how many elements in an array are negative.

■■ **R7.8** Rewrite the following loops without using the enhanced `for` loop construct. Here, `values` is an array of floating-point numbers.

a.
```
for (double x : values) { total = total + x; }
```

b.
```
for (double x : values) { if (x == target) { return true; } }
```

c.
```
int i = 0;
for (double x : values) { values[i] = 2 * x; i++; }
```

■■ **R7.9** Rewrite the following loops, using the enhanced `for` loop construct. Here, `values` is an array of floating-point numbers.

a.
```
for (int i = 0; i < values.length; i++) { total = total + values[i]; }
```

b.
```
for (int i = 1; i < values.length; i++) { total = total + values[i]; }
```

c.
```
for (int i = 0; i < values.length; i++)
{
   if (values[i] == target) { return i; }
}
```

■ **R7.10** What is wrong with each of the following code segments?

a.
```
ArrayList<int> values = new ArrayList<int>();
```

b.
```
ArrayList<Integer> values = new ArrayList();
```

c.
```
ArrayList<Integer> values = new ArrayList<Integer>;
```

d.
```
ArrayList<Integer> values = new ArrayList<Integer>();
for (int i = 1; i <= 10; i++)
{
   values.set(i - 1, i * i);
}
```

e.
```
ArrayList<Integer> values;
for (int i = 1; i <= 10; i++)
{
   values.add(i * i);
}
```

■ **R7.11** What is an index of an array? What are the legal index values? What is a bounds error?

■ **R7.12** Write a program that contains a bounds error. Run the program. What happens on your computer?

■ **R7.13** Write a loop that reads ten numbers and a second loop that displays them in the opposite order from which they were entered.

■■ **R7.14** For the operations on partially filled arrays below, provide the header of a method. Do not implement the methods.

a. Sort the elements in decreasing order.

b. Print all elements, separated by a given string.

c. Count how many elements are less than a given value.

d. Remove all elements that are less than a given value.

e. Place all elements that are less than a given value in another array.

■ **R7.15** Trace the flow of the loop in Section 7.3.4 with the given example. Show two columns, one with the value of `i` and one with the output.

■ **R7.16** Consider the following loop for collecting all elements that match a condition; in this case, that the element is larger than 100.

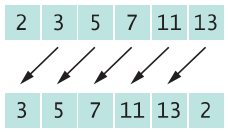
```
ArrayList<Double> matches = new ArrayList<Double>();
for (double element : values)
{
   if (element > 100)
   {
      matches.add(element);
   }
}
```

Trace the flow of the loop, where `values` contains the elements 110 90 100 120 80. Show two columns, for `element` and `matches`.

■ **R7.17** Trace the flow of the loop in Section 7.3.5, where `values` contains the elements 80 90 100 120 110. Show two columns, for `pos` and `found`. Repeat the trace when `values` contains the elements 80 90 120 70.

■■ **R7.18** Trace the algorithm for removing an element described in Section 7.3.6. Use an array `values` with elements 110 90 100 120 80, and remove the element at index 2.

■■ **R7.19** Give pseudocode for an algorithm that rotates the elements of an array by one position, moving the initial element to the end of the array, like this:



■■ **R7.20** Give pseudocode for an algorithm that removes all negative values from an array, preserving the order of the remaining elements.

■■ **R7.21** Suppose `values` is a *sorted* array of integers. Give pseudocode that describes how a new value can be inserted in its proper position so that the resulting array stays sorted.

■■■ **R7.22** A *run* is a sequence of adjacent repeated values. Give pseudocode for computing the length of the longest run in an array. For example, the longest run in the array with elements

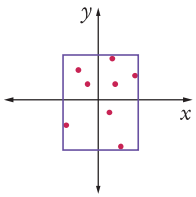
```
1 2 5 5 3 1 2 4 3 2 2 2 2 3 6 5 5 6 3 1
```

has length 4.

■■■ **R7.23** What is wrong with the following method that aims to fill an array with random numbers?

```
public void makeCombination(int[] values, int n)
{
   Random generator = new Random();
   int[] numbers = new int[values.length];
   for (int i = 0; i < numbers.length; i++)
   {
      numbers[i] = generator.nextInt(n);
   }
   values = numbers;
}
```

■■ **R7.24** You are given two arrays denoting x- and y-coordinates of a set of points in the plane. For plotting the point set, we need to know the x- and y-coordinates of the smallest rectangle containing the points.



How can you obtain these values from the fundamental algorithms in Section 7.3?

■ **R7.25** Solve the problem described in Section 7.4 by sorting the array first. How do you need to modify the algorithm for computing the total?

■■ **R7.26** Solve the task described in Section 7.5 using an algorithm that removes and inserts elements instead of switching them. Write the pseudocode for the algorithm, assuming that methods for removal and insertion exist. Act out the algorithm with a sequence of coins and explain why it is less efficient than the swapping algorithm developed in Section 7.5.

■■ **R7.27** Develop an algorithm for finding the most frequently occurring value in an array of numbers. Use a sequence of coins. Place paper clips below each coin that count how many other coins of the same value are in the sequence. Give the pseudocode for an algorithm that yields the correct answer, and describe how using the coins and paper clips helped you find the algorithm.

■■ **R7.28** Write Java statements for performing the following tasks with an array declared as

```
int[][] values = new int[ROWS][COLUMNS];
```

- Fill all entries with 0.
- Fill elements alternately with 0s and 1s in a checkerboard pattern.
- Fill only the elements in the top and bottom rows with zeroes.
- Compute the sum of all elements.
- Print the array in tabular form.

■■ **R7.29** Write pseudocode for an algorithm that fills the first and last columns as well as the first and last rows of a two-dimensional array of integers with –1.

■ **R7.30** Section 7.7.7 shows that you must be careful about updating the index value when you remove elements from an array list. Show how you can avoid this problem by traversing the array list backwards.

■■ **R7.31** True or false?

a. All elements of an array are of the same type.

b. Arrays cannot contain strings as elements.

c. Two-dimensional arrays always have the same number of rows and columns.

d. Elements of different columns in a two-dimensional array can have different types.

e. A method cannot return a two-dimensional array.

f. A method cannot change the length of an array argument.

g. A method cannot change the number of columns of an argument that is a two-dimensional array.

■■ **R7.32** How do you perform the following tasks with array lists in Java?

a. Test that two array lists contain the same elements in the same order.

b. Copy one array list to another.

c. Fill an array list with zeroes, overwriting all elements in it.

d. Remove all elements from an array list.

■ **R7.33** True or false?

a. All elements of an array list are of the same type.

b. Array list index values must be integers.

c. Array lists cannot contain strings as elements.

d. Array lists can change their size, getting larger or smaller.

e. A method cannot return an array list.

f. A method cannot change the size of an array list argument.

■ **Testing R7.34** Define the terms regression testing and test suite.

■■ **Testing R7.35** What is the debugging phenomenon known as *cycling*? What can you do to avoid it?

PRACTICE EXERCISES

■■ **E7.1** Write a program that initializes an array with ten random integers and then prints four lines of output, containing

- Every element at an even index.
- Every even element.
- All elements in reverse order.
- Only the first and last element.

■■ **E7.2** Write array methods that carry out the following tasks for an array of integers by completing the `ArrayMethods` class below. For each method, provide a test program.

```
public class ArrayMethods
{
   private int[] values;
   public ArrayMethods(int[] initialValues) { values = initialValues; }
   public void swapFirstAndLast() { ... }
   public void shiftRight() { ... }
   ...
}
```

a. Swap the first and last elements in the array.

b. Shift all elements by one to the right and move the last element into the first position. For example, 1 4 9 16 25 would be transformed into 25 1 4 9 16.

c. Replace all even elements with 0.

d. Replace each element except the first and last by the larger of its two neighbors.

e. Remove the middle element if the array length is odd, or the middle two elements if the length is even.

f. Move all even elements to the front, otherwise preserving the order of the elements.

g. Return the second-largest element in the array.

h. Return true if the array is currently sorted in increasing order.

i. Return true if the array contains two adjacent duplicate elements.

j. Return true if the array contains duplicate elements (which need not be adjacent).

■ **E7.3** Modify the `LargestInArray.java` program in Section 7.3 to mark both the smallest and the largest elements.

■■ **E7.4** Write a method `sumWithoutSmallest` that computes the sum of an array of values, except for the smallest one, in a single loop. In the loop, update the sum and the smallest value. After the loop, return the difference.

■ **E7.5** Add a method `removeMin` to the `Student` class of Section 7.4 that removes the minimum score without calling other methods.

■■ **E7.6** Compute the *alternating sum* of all elements in an array. For example, if your program reads the input

1 4 9 16 9 7 4 9 11

then it computes

$$1 - 4 + 9 - 16 + 9 - 7 + 4 - 9 + 11 = -2$$

■ **E7.7** Write a method that reverses the sequence of elements in an array. For example, if you call the method with the array

1 4 9 16 9 7 4 9 11

then the array is changed to

11 9 4 7 9 16 9 4 1

■■■ **E7.8** Write a program that produces ten random permutations of the numbers 1 to 10. To generate a random permutation, you need to fill an array with the numbers 1 to 10 so that no two entries of the array have the same contents. You could do it by brute force, by generating random values until you have a value that is not yet in the array. But that is inefficient. Instead, follow this algorithm.

```
Make a second array and fill it with the numbers 1 to 10.
Repeat 10 times
   Pick a random element from the second array.
   Remove it and append it to the permutation array.
```

■ **E7.9** Write a method that implements the algorithm developed in Section 7.5.

■■ **E7.10** Consider the following class:

```
public class Sequence
{
    private int[] values;
    public Sequence(int size) { values = new int[size]; }
    public void set(int i, int n) { values[i] = n; }
}
```

Add a method

```
public boolean equals(Sequence other)
```

that checks whether the two sequences have the same values in the same order.

■■ **E7.11** Add a method

```
public boolean sameValues(Sequence other)
```

to the `Sequence` class of Exercise E7.10 that checks whether two sequences have the same values in some order, ignoring duplicates. For example, the two sequences

1 4 9 16 9 7 4 9 11

and

11 11 7 9 16 4 1

would be considered identical. You will probably need one or more helper methods.

■■■ **E7.12** Add a method

```
public boolean isPermutationOf(Sequence other)
```

to the `Sequence` class of Exercise E7.10 that checks whether two sequences have the same values in some order, with the same multiplicities. For example,

1 4 9 16 9 7 4 9 11

is a permutation of

11 1 4 9 16 9 7 4 9

but

1 4 9 16 9 7 4 9 11

is not a permutation of

11 11 7 9 16 4 1 4 9

You will probably need one or more helper methods.

■■ **E7.13** Write a program that generates a sequence of 20 random values between 0 and 99 in an array, prints the sequence, sorts it, and prints the sorted sequence. Use the `sort` method from the standard Java library.

■■ **E7.14** Consider the following class:

```
public class Table
{
    private int[][] values;
    public Table(int rows, int columns) { values = new int[rows][columns]; }
    public void set(int i, int j, int n) { values[i][j] = n; }
}
```

Add a method that computes the average of the neighbors of a table element in the eight directions shown in Figure 15.

```
public double neighborAverage(int row, int column)
```

However, if the element is located at the boundary of the array, only include the neighbors that are in the table. For example, if `row` and `column` are both 0, there are only three neighbors.

■■■ **E7.15** *Magic squares*. An $n \times n$ matrix that is filled with the numbers $1, 2, 3, \ldots, n^2$ is a magic square if the sum of the elements in each row, in each column, and in the two diagonals is the same value.

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

Write a program that reads in 16 values from the keyboard and tests whether they form a magic square when put into a 4 × 4 array.

You need to test two features:

1. Does each of the numbers 1, 2, ..., 16 occur in the user input?
2. When the numbers are put into a square, are the sums of the rows, columns, and diagonals equal to each other?

■■■ **E7.16** Implement the following algorithm to construct magic $n \times n$ squares; it works only if n is odd.

```
Set row = n - 1, column = n / 2.
For k = 1 ... n * n
   Place k at [row][column].
   Increment row and column.
   If the row or column is n, replace it with 0.
   If the element at [row][column] has already been filled
      Set row and column to their previous values.
      Decrement row.
```

Here is the 5 × 5 square that you get if you follow this method:

11	18	25	2	9
10	12	19	21	3
4	6	13	20	22
23	5	7	14	16
17	24	1	8	15

Write a program whose input is the number n and whose output is the magic square of order n if n is odd.

■■ **E7.17** Write a program that reads a sequence of input values and displays a bar chart of the values, using asterisks, like this:

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

You may assume that all values are positive. First figure out the maximum value. That value's bar should be drawn with 40 asterisks. Shorter bars should use proportionally fewer asterisks.

■■■ **E7.18** Improve the program of Exercise E7.17 to work correctly when the data set contains negative values.

■■ **E7.19** Improve the program of Exercise E7.17 by adding captions for each bar. Prompt the user for the captions and data values. The output should look like this:

```
      Egypt **********************
     France ****************************************
      Japan ****************************
    Uruguay *************************
Switzerland **************
```

■ **E7.20** Consider the following class:

```
public class Sequence
{
    private ArrayList<Integer> values;
    public Sequence() { values = new ArrayList<Integer>(); }
    public void add(int n) { values.add(n); }
    public String toString() { return values.toString(); }
}
```

Add a method

```
public Sequence append(Sequence other)
```

that creates a new sequence, appending this and the other sequence, without modifying either sequence. For example, if `a` is

1 4 9 16

and `b` is the sequence

9 7 4 9 11

then the call `a.append(b)` returns the sequence

1 4 9 16 9 7 4 9 11

without modifying `a` or `b`.

■■ **E7.21** Add a method

```
public Sequence merge(Sequence other)
```

to the `Sequence` class of Exercise E7.20 that merges two sequences, alternating elements from both sequences. If one sequence is shorter than the other, then alternate as long as you can and then append the remaining elements from the longer sequence. For example, if `a` is

1 4 9 16

and `b` is

9 7 4 9 11

then `a.merge(b)` returns the sequence

1 9 4 7 9 4 16 9 11

without modifying `a` or `b`.

■■ **E7.22** Add a method

```
public Sequence mergeSorted(Sequence other)
```

to the `Sequence` class of Exercise E7.20 that merges two sorted sequences, producing a new sorted sequence. Keep an index into each sequence, indicating how much of it has been processed already. Each time, append the smallest unprocessed value from either sequence, then advance the index. For example, if `a` is

1 4 9 16

and `b` is

4 7 9 9 11

then `a.mergeSorted(b)` returns the sequence

1 4 4 7 9 9 9 11 16

If `a` or `b` is not sorted, merge the longest prefixes of `a` and `b` that are sorted.

PROGRAMMING PROJECTS

■■ **P7.1** A *run* is a sequence of adjacent repeated values. Write a program that generates a sequence of 20 random die tosses in an array and that prints the die values, marking the runs by including them in parentheses, like this:

```
1 2 (5 5) 3 1 2 4 3 (2 2 2 2) 3 6 (5 5) 6 3 1
```

Use the following pseudocode:

```
Set a boolean variable inRun to false.
For each valid index i in the array
   If inRun
      If values[i] is different from the preceding value
         Print ).
         inRun = false.
   If not inRun
      If values[i] is the same as the following value
         Print (.
         inRun = true.
   Print values[i].
If inRun, print ).
```

■■ **P7.2** Write a program that generates a sequence of 20 random die tosses in an array and that prints the die values, marking only the longest run, like this:

```
1 2 5 5 3 1 2 4 3 (2 2 2 2) 3 6 5 5 6 3 1
```

If there is more than one run of maximum length, mark the first one.

■■ **P7.3** It is a well-researched fact that men in a restroom generally prefer to maximize their distance from already occupied stalls, by occupying the middle of the longest sequence of unoccupied places.

For example, consider the situation where ten stalls are empty.

```
_ _ _ _ _ _ _ _ _ _
```

The first visitor will occupy a middle position:

```
_ _ _ _ _ X _ _ _ _
```

The next visitor will be in the middle of the empty area at the left.

```
_ _ X _ _ X _ _ _ _
```

Write a program that reads the number of stalls and then prints out diagrams in the format given above when the stalls become filled, one at a time. *Hint:* Use an array of `boolean` values to indicate whether a stall is occupied.

■■■ **P7.4** In this assignment, you will model the game of *Bulgarian Solitaire*. The game starts with 45 cards. (They need not be playing cards. Unmarked index cards work just as well.) Randomly divide them into some number of piles of random size. For example, you might start with piles of size 20, 5, 1, 9, and 10. In each round, you take one card from each pile, forming a new pile with these cards. For example, the sample starting configuration would be transformed into piles of size 19, 4, 8, 9, and 5. The solitaire is over when the piles have size 1, 2, 3, 4, 5, 6, 7, 8, and 9, in some order. (It can be shown that you always end up with such a configuration.)

In your program, produce a random starting configuration and print it. Then keep applying the solitaire step and print the result. Stop when the solitaire final configuration is reached.

■■ **P7.5** A theater seating chart is implemented as a two-dimensional array of ticket prices, like this:

```
10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10
10 10 10 10 10 10 10 10 10 10
10 10 20 20 20 20 20 20 10 10
10 10 20 20 20 20 20 20 10 10
10 10 20 20 20 20 20 20 10 10
20 20 30 30 40 40 30 30 20 20
20 30 30 40 50 50 40 30 30 20
30 40 50 50 50 50 50 50 40 30
```

Write a program that prompts users to pick either a seat or a price. Mark sold seats by changing the price to 0. When a user specifies a seat, make sure it is available. When a user specifies a price, find any seat with that price.

■■■ **P7.6** Write a program that plays tic-tac-toe. The tic-tac-toe game is played on a 3 × 3 grid as in the photo at right. The game is played by two players, who take turns. The first player marks moves with a circle, the second with a cross. The player who has formed a horizontal, vertical, or diagonal sequence of three marks wins. Your program should draw the game board, ask the user for the coordinates of the next mark, change the players after every successful move, and pronounce the winner.

■■■ **P7.7** In this assignment, you will implement a simulation of a popular casino game usually called video poker. The card deck contains 52 cards, 13 of each suit. At the beginning of the game, the deck is shuffled. You need to devise a fair method for shuffling. (It does not have to be efficient.) The player pays a token for each game. Then the top five cards of the deck are presented to the player. The player can reject none, some, or all of the cards. The rejected cards are replaced from the top of the deck. Now the hand is scored. Your program should pronounce it to be one of the following:

- No pair—The lowest hand, containing five separate cards that do not match up to create any of the hands below.
- One pair—Two cards of the same value, for example two queens. Payout: 1
- Two pairs—Two pairs, for example two queens and two 5's. Payout: 2
- Three of a kind—Three cards of the same value, for example three queens. Payout: 3
- Straight—Five cards with consecutive values, not necessarily of the same suit, such as 4, 5, 6, 7, and 8. The ace can either precede a 2 or follow a king. Payout: 4
- Flush—Five cards, not necessarily in order, of the same suit. Payout: 5
- Full House—Three of a kind and a pair, for example three queens and two 5's. Payout: 6
- Four of a Kind—Four cards of the same value, such as four queens. Payout: 25
- Straight Flush—A straight and a flush: Five cards with consecutive values of the same suit. Payout: 50
- Royal Flush—The best possible hand in poker. A 10, jack, queen, king, and ace, all of the same suit. Payout: 250

■■■ **P7.8** *The Game of Life* is a well-known mathematical game that gives rise to amazingly complex behavior, although it can be specified by a few simple rules. (It is not actually a game in the traditional sense, with players competing for a win.) Here are the rules. The game is played on a rectangular board. Each square can be either empty or occupied. At the beginning, you can specify empty and occupied cells in some way; then the game runs automatically. In each *generation*, the next generation is computed. A new cell is born on an empty square if it is surrounded by exactly three occupied neighbor cells. A cell dies of overcrowding if it is surrounded by four or more neighbors, and it dies of loneliness if it is surrounded by zero or one neighbor. A neighbor is an occupant of an adjacent square to the left, right, top, or bottom or in a diagonal direction. Figure 21 shows a cell and its neighbor cells.

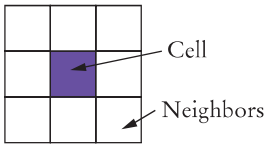


Figure 21
Neighborhood of a Cell

Many configurations show interesting behavior when subjected to these rules. Figure 22 shows a *glider*, observed over five generations. After four generations, it is transformed into the identical shape, but located one square to the right and below.

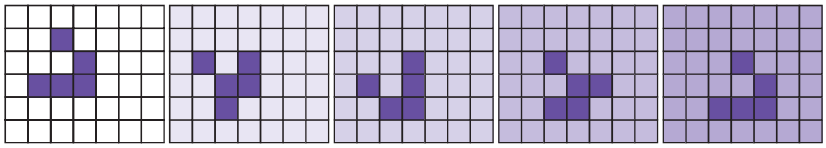


Figure 22 Glider

One of the more amazing configurations is the glider gun: a complex collection of cells that, after 30 moves, turns back into itself and a glider (see Figure 23).

Program the game to eliminate the drudgery of computing successive generations by hand. Use a two-dimensional array to store the rectangular configuration. Write a program that shows successive generations of the game. Ask the user to specify the original configuration, by typing in a configuration of spaces and o characters.

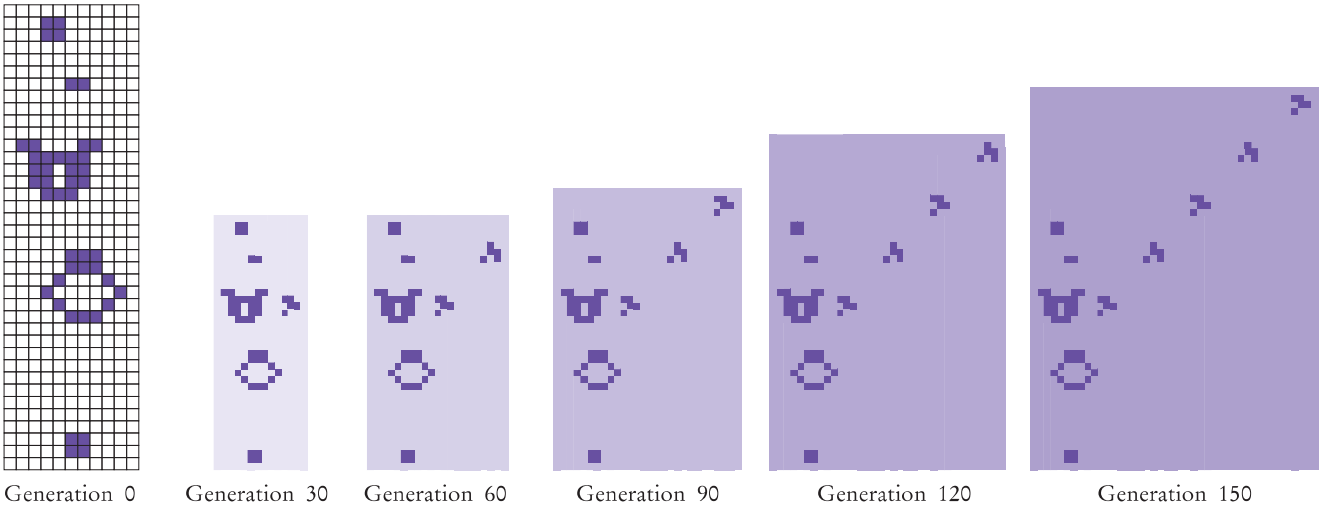


Figure 23 Glider Gun

■■ Business P7.9 A pet shop wants to give a discount to its clients if they buy one or more pets and at least five other items. The discount is equal to 20 percent of the cost of the other items, but not the pets.

Use a class `Item` to describe an item, with any needed methods and a constructor

```
public Item(double price, boolean isPet, int quantity)
```

An invoice holds a collection of `Item` objects; use an array or array list to store them. In the `Invoice` class, implement methods

```
public void add(Item anItem)
public double getDiscount()
```

Write a program that prompts a cashier to enter each price and quantity, and then a `Y` for a pet or `N` for another item. Use a price of –1 as a sentinel. In the loop, call the `add` method; after the loop, call the `getDiscount` method and display the returned value.

■■ Business P7.10 A supermarket wants to reward its best customer of each day, showing the customer's name on a screen in the supermarket. For that purpose, the store keeps an `ArrayList<Customer>`. In the `Store` class, implement methods

```
public void addSale(String customerName, double amount)
public String nameOfBestCustomer()
```

to record the sale and return the name of the customer with the largest sale.

Write a program that prompts the cashier to enter all prices and names, adds them to a `Store` object, and displays the best customer's name. Use a price of `0` as a sentinel.

■■■ Business P7.11 Improve the program of Exercise P7.10 so that it displays the top customers, that is, the `topN` customers with the largest sales, where `topN` is a value that the user of the program supplies. Implement a method

```
public ArrayList<String> nameOfBestCustomers(int topN)
```

If there were fewer than `topN` customers, include all of them.

■■ Science P7.12 Sounds can be represented by an array of "sample values" that describe the intensity of the sound at a point in time. The program in `ch07/sound` of your companion code reads a sound file (in WAV format), processes the sample values, and shows the result. Your task is to process the sound by introducing an echo. For each sound value, add the value from 0.2 seconds ago. Scale the result so that no value is larger than 32767.

■■■ Science P7.13 You are given a two-dimensional array of values that give the height of a terrain at different points in a square. Write a constructor

```
public Terrain(double[][] heights)
```

and a method

```
public void printFloodMap(double waterLevel)
```

that prints out a flood map, showing which of the points in the terrain would be flooded if the water level was the given value.

In the flood map, print a * for each flooded point and a space for each point that is not flooded.

Here is a sample map:

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

Then write a program that reads one hundred terrain height values and shows how the terrain gets flooded when the water level increases in ten steps from the lowest point in the terrain to the highest.

■■ Science P7.14 Sample values from an experiment often need to be smoothed out. One simple approach is to replace each value in an array with the average of the value and its two neighboring values (or one neighboring value if it is at either end of the array). Given a class `Data` with instance fields

```
private double[] values;
private double valuesSize;
```

implement a method

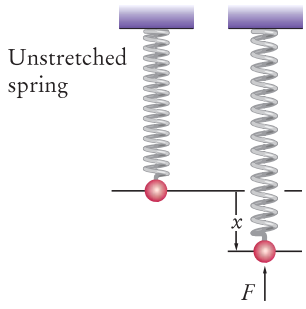
```
public void smooth()
```

that carries out this operation. You should not create another array in your solution.

■■■ Science P7.15 Write a program that models the movement of an object with mass m that is attached to an oscillating spring. When a spring is displaced from its equilibrium position by an amount x, Hooke's law states that the restoring force is

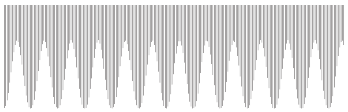
$$F = -kx$$

where k is a constant that depends on the spring. (Use 10 N/m for this simulation.)

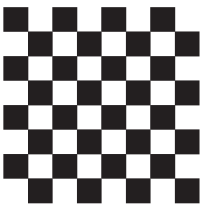


Start with a given displacement x (say, 0.5 meter). Set the initial velocity v to 0. Compute the acceleration a from Newton's law ($F = ma$) and Hooke's law, using a mass of 1 kg. Use a small time interval $\Delta t = 0.01$ second. Update the velocity—it changes by $a\Delta t$. Update the displacement—it changes by $v\Delta t$.

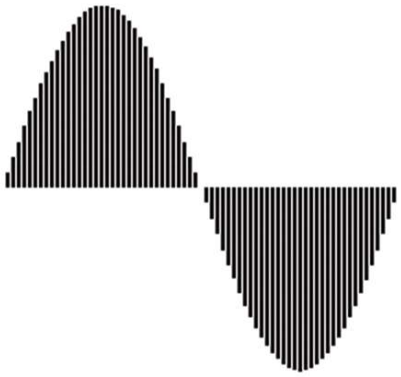
Every ten iterations, plot the spring displacement as a bar, where 1 pixel represents 1 cm, as shown here.



■■ Graphics P7.16 Generate the image of a checkerboard.



■ **Graphics P7.17** Generate the image of a sine wave. Draw a line of pixels for every five degrees.



■ **Graphics P7.18** Implement a class `Cloud` that contains an array list of `Point2D.Double` objects. Support methods

```
public void add(Point2D.Double aPoint)
public void draw(Graphics2D g2)
```

Draw each point as a tiny circle. Write a graphical application that draws a cloud of 100 random points.

■■ **Graphics P7.19** Implement a class `Polygon` that contains an array list of `Point2D.Double` objects. Support methods

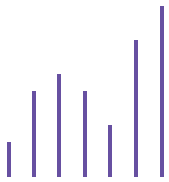
```
public void add(Point2D.Double aPoint)
public void draw(Graphics2D g2)
```

Draw the polygon by joining adjacent points with a line, and then closing it up by joining the end and start points. Write a graphical application that draws a square and a pentagon using two `Polygon` objects.

■ **Graphics P7.20** Write a class `Chart` with methods

```
public void add(int value)
public void draw(Graphics2D g2)
```

that displays a stick chart of the added values, like this:



You may assume that the values are pixel positions.

■■ **Graphics P7.21** Write a class `BarChart` with methods

```
public void add(double value)
public void draw(Graphics2D g2)
```

that displays a bar chart of the added values. You may assume that all added values are positive. Stretch the bars so that they fill the entire area of the screen. You must figure out the maximum of the values, then scale each bar.

■■■ **Graphics P7.22** Improve the `BarChart` class of Exercise P7.21 to work correctly when the data contains negative values.

■■ **Graphics P7.23** Write a class `PieChart` with methods

```
public void add(double value)
public void draw(Graphics2D g2)
```

that displays a pie chart of the added values. Assume that all data values are positive.

ANSWERS TO SELF-CHECK QUESTIONS

1. `int[] primes = { 2, 3, 5, 7, 11 };`
2. `2, 3, 5, 3, 2`
3. `3, 4, 6, 8, 12`
4. `values[0] = 10;`
   `values[9] = 10;` or better:
   `values[values.length - 1] = 10;`
5. `String[] words = new String[10];`
6. `String[] words = { "Yes", "No" };`
7. No. Because you don't store the values, you need to print them when you read them. But you don't know where to add the `<=` until you have seen all values.
8.
```
public class Lottery
{
   public int[] getCombination(int n) { . . . }
   . . .
}
```
9. It counts how many elements of `values` are zero.
10.
```
for (double x : values)
{
   System.out.println(x);
}
```
11.
```
double product = 1;
for (double f : factors)
{
   product = product * f;
}
```
12. The loop writes a value into `values[i]`. The enhanced `for` loop does not have the index variable `i`.
13.
```
20 <== largest value
10
20 <== largest value
```
14.
```
int count = 0;
for (double x : values)
{
   if (x == 0) { count++; }
}
```
15. If all elements of `values` are negative, then the result is incorrectly computed as `0`.
16.
```
for (int i = 0; i < values.length; i++)
{
   System.out.print(values[i]);
   if (i < values.length - 1)
   {
      System.out.print(" | ");
   }
}
```
    Now you know why we set up the loop the other way.
17. If the array has no elements, then the program terminates with an exception.
18. If there is a match, then `pos` is incremented before the loop exits.
19. This loop sets all elements to `values[pos]`.
20. Use the first algorithm. The order of elements does not matter when computing the sum.
21. **Find the minimum value.**
    Calculate the sum.
    Subtract the minimum value.
22. Use the algorithm for counting matches (Section 6.7.2) twice, once for counting the positive values and once for counting the negative values.
23. You need to modify the algorithm in Section 7.3.4.
```
boolean first = true;
for (int i = 0; i < values.length; i++)
{
   if (values[i] > 0))
   {
      if (first) { first = false; }
      else { System.out.print(", "); }
   }
   System.out.print(values[i]);
}
```
    Note that you can no longer use `i > 0` as the criterion for printing a separator.
24. Use the algorithm to collect all positive elements in an array, then use the algorithm in Section 7.3.4 to print the array of matches.
25. The paperclip for `i` assumes positions 0, 1, 2, 3. When `i` is incremented to 4, the condition `i < size / 2` becomes false, and the loop ends. Similarly, the paperclip for `j` assumes positions 4, 5, 6, 7, which are the valid positions for the second half of the array.

26. It reverses the elements in the array.

27. Here is one solution. The basic idea is to move all odd elements to the end. Put one paper clip at the beginning of the array and one at the end. If the element at the first paper clip is odd, swap it with the one at the other paper clip and move that paper clip to the left. Otherwise, move the first paper clip to the right. Stop when the two paper clips meet. Here is the pseudocode:

```
i = 0
j = size - 1
While (i < j)
   If (a[i] is odd)
      Swap elements at positions i and j.
      j--
   Else
      i++
```

28. Here is one solution. The idea is to remove all odd elements and move them to the end. The trick is to know when to stop. Nothing is gained by moving odd elements into the area that already contains moved elements, so we want to mark that area with another paper clip.

```
i = 0
moved = size
While (i < moved)
   If (a[i] is odd)
      Remove the element at position i and add it
         at the end.
      moved--
```

29. When you read inputs, you get to see values one at a time, and you can't peek ahead. Picking cards one at a time from a deck of cards simulates this process better than looking at a sequence of items, all of which are revealed.

30. You get the total number of gold, silver, and bronze medals in the competition. In our example, there are four of each.

31.
```
for (int i = 0; i < 8; i++)
{
   for (int j = 0; j < 8; j++)
   {
      board[i][j] = (i + j) % 2;
   }
}
```

32. `String[][] board = new String[3][3];`

33. `board[0][2] = "x";`

34. `board[0][0]`, `board[1][1]`, `board[2][2]`

35.
```
ArrayList<Integer> primes =
   new ArrayList<Integer>();
primes.add(2);
primes.add(3);
primes.add(5);
primes.add(7);
primes.add(11);
```

36.
```
for (int i = primes.size() - 1; i >= 0; i--)
{
   System.out.println(primes.get(i));
}
```

37. `"Ann"`, `"Cal"`

38. The `names` variable has not been initialized.

39. `names1` contains `"Emily"`, `"Bob"`, `"Cindy"`, `"Dave"`; `names2` contains `"Dave"`

40. Because the number of weekdays doesn't change, there is no disadvantage to using an array, and it is easier to initialize:

```
String[] weekdayNames = { "Monday", "Tuesday",
   "Wednesday", "Thursday", "Friday",
   "Saturday", "Sunday" };
```

41. Reading inputs into an array list is much easier.

42. It is possible to introduce errors when modifying code.

43. Add a test case to the test suite that verifies that the error is fixed.

44. There is no human user who would see the prompts because input is provided from a file.

CHAPTER 8

DESIGNING CLASSES

CHAPTER GOALS

To learn how to choose appropriate classes for a given problem

To understand the concept of cohesion

To minimize dependencies and side effects

To learn how to find a data representation for a class

To understand static methods and variables

To learn about packages

To learn about unit testing frameworks

CHAPTER CONTENTS

Good design should be both functional and attractive. When designing classes, each class should be dedicated to a particular purpose, and classes should work well together. In this chapter, you will learn how to discover classes, design good methods, and choose appropriate data representations. You will also learn how to design features that belong to the class as a whole, not individual objects, by using static methods and variables. You will see how to use packages to organize your classes. Finally, we introduce the JUnit testing framework that lets you verify the functionality of your classes.

8.1 Discovering Classes

You have used a good number of classes in the preceding chapters and probably designed a few classes yourself as part of your programming assignments. Designing a class can be a challenge—it is not always easy to tell how to start or whether the result is of good quality.

A class should represent a single concept from a problem domain, such as business, science, or mathematics.

What makes a good class? Most importantly, a class should *represent a single concept* from a problem domain. Some of the classes that you have seen represent concepts from mathematics:

- `Point`
- `Rectangle`
- `Ellipse`

Other classes are abstractions of real-life entities:

- `BankAccount`
- `CashRegister`

For these classes, the properties of a typical object are easy to understand. A `Rectangle` object has a width and height. Given a `BankAccount` object, you can deposit and withdraw money. Generally, concepts from a domain related to the program's purpose, such as science, business, or gaming, make good classes. The name for such a class should be a noun that describes the concept. In fact, a simple rule of thumb for getting started with class design is to look for nouns in the problem description.

One useful category of classes can be described as *actors*. Objects of an actor class carry out certain tasks for you. Examples of actors are the `Scanner` class of Chapter 4 and the `Random` class in Chapter 6. A `Scanner` object scans a stream for numbers and strings. A `Random` object generates random numbers. It is a good idea to choose class names for actors that end in "-er" or "-or". (A better name for the `Random` class might be `RandomNumberGenerator`.)

Very occasionally, a class has no objects, but it contains a collection of related static methods and constants. The `Math` class is an example. Such a class is called a *utility class*.

Finally, you have seen classes with only a `main` method. Their sole purpose is to start a program. From a design perspective, these are somewhat degenerate examples of classes.

What might not be a good class? If you can't tell from the class name what an object of the class is supposed to do, then you are probably not on the right track. For

example, your homework assignment might ask you to write a program that prints paychecks. Suppose you start by trying to design a class `PaycheckProgram`. What would an object of this class do? An object of this class would have to do everything that the homework needs to do. That doesn't simplify anything. A better class would be `Paycheck`. Then your program can manipulate one or more `Paycheck` objects.

Another common mistake is to turn a single operation into a class. For example, if your homework assignment is to compute a paycheck, you may consider writing a class `ComputePaycheck`. But can you visualize a "ComputePaycheck" object? The fact that "ComputePaycheck" isn't a noun tips you off that you are on the wrong track. On the other hand, a `Paycheck` class makes intuitive sense. The word "paycheck" is a noun. You can visualize a paycheck object. You can then think about useful methods of the `Paycheck` class, such as `computeTaxes`, that help you solve the assignment.

1. What is a simple rule of thumb for finding classes?
2. Your job is to write a program that plays chess. Might `ChessBoard` be an appropriate class? How about `MovePiece`?

Practice It Now you can try these exercises at the end of the chapter: R8.1, R8.2, R8.3.

8.2 Designing Good Methods

In the following sections, you will learn several useful criteria for analyzing and improving the public interface of a class.

8.2.1 Providing a Cohesive Public Interface

The public interface of a class is cohesive if all of its features are related to the concept that the class represents.

A class should represent a single concept. All interface features should be closely related to the single concept that the class represents. Such a public interface is said to be **cohesive**.

The members of a cohesive team have a common goal.

If you find that the public interface of a class refers to multiple concepts, then that is a good sign that it may be time to use separate classes instead. Consider, for example, the public interface of the `CashRegister` class in Chapter 4:

```
public class CashRegister
{
   public static final double QUARTER_VALUE = 0.25;
   public static final double DIME_VALUE = 0.1;
   public static final double NICKEL_VALUE = 0.05;
   . . .
   public void receivePayment(int dollars, int quarters,
         int dimes, int nickels, int pennies)
   . . .
}
```

There are really two concepts here: a cash register that holds coins and computes their total, and the values of individual coins. (For simplicity, we assume that the cash register only holds coins, not bills. Exercise E8.3 discusses a more general solution.)

It makes sense to have a separate `Coin` class and have coins responsible for knowing their values.

```
public class Coin
{
   . . .
   public Coin(double aValue, String aName) { . . . }
   public double getValue() { . . . }
   . . .
}
```

Then the `CashRegister` class can be simplified:

```
public class CashRegister
{
   . . .
   public void receivePayment(int coinCount, Coin coinType) { . . . }
   {
      payment = payment + coinCount * coinType.getValue();
   }
   . . .
}
```

Now the `CashRegister` class no longer needs to know anything about coin values. The same class can equally well handle euros or zorkmids!

This is clearly a better solution, because it separates the responsibilities of the cash register and the coins. The only reason we didn't follow this approach in Chapter 4 was to keep the `CashRegister` example simple.

8.2.2 Minimizing Dependencies

A class depends on another class if its methods use that class in any way.

Many methods need other classes in order to do their jobs. For example, the `receivePayment` method of the restructured `CashRegister` class now uses the `Coin` class. We say that the `CashRegister` class *depends on* the `Coin` class.

To visualize relationships between classes, such as dependence, programmers draw class diagrams. In this book, we use the UML ("**Unified Modeling Language**") notation for objects and classes. UML is a notation for object-oriented analysis and design invented by Grady Booch, Ivar Jacobson, and James Rumbaugh, three leading researchers in object-oriented softare development. (Appendix H has a summary of the UML notation used in this book.) The UML notation distinguishes between *object diagrams* and class diagrams. In an object diagram the class names are underlined; in a class diagram the class names are not underlined. In a class diagram, you denote dependency by a dashed line with a ≫-shaped open arrow tip that points to the dependent class. Figure 1 shows a class diagram indicating that the `CashRegister` class depends on the `Coin` class.

Note that the `Coin` class does *not* depend on the `CashRegister` class. All `Coin` methods can carry out their work without ever calling any method in the `CashRegister` class. Conceptually, coins have no idea that they are being collected in cash registers.

Here is an example of minimizing dependencies. Consider how we have always printed a bank balance:

```
System.out.println("The balance is now $" + momsSavings.getBalance());
```

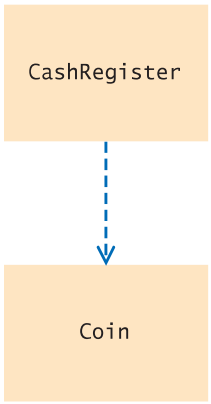


Figure 1
Dependency Relationship Between the `CashRegister` and `Coin` Classes

Why don't we simply have a `printBalance` method?

```
public void printBalance() // Not recommended
{
   System.out.println("The balance is now $" + balance);
}
```

The method depends on `System.out`. Not every computing environment has `System.out`. For example, an automatic teller machine doesn't display console messages. In other words, this design violates the rule of minimizing dependencies. The `printBalance` method couples the `BankAccount` class with the `System` and `PrintStream` classes.

It is best to place the code for producing output or consuming input in a separate class. That way, you decouple input/output from the actual work of your classes.

8.2.3 Separating Accessors and Mutators

A **mutator method** changes the state of an object. Conversely, an **accessor method** asks an object to compute a result, without changing the state.

An immutable class has no mutator methods.

Some classes have been designed to have only accessor methods and no mutator methods at all. Such classes are called **immutable**. An example is the `String` class. Once a string has been constructed, its content never changes. No method in the `String` class can modify the contents of a string. For example, the `toUpperCase` method does not change characters from the original string. Instead, it constructs a *new* string that contains the uppercase characters:

```
String name = "John Q. Public";
String uppercased = name.toUpperCase(); // name is not changed
```

References to objects of an immutable class can be safely shared.

An immutable class has a major advantage: It is safe to give out references to its objects freely. If no method can change the object's value, then no code can modify the object at an unexpected time.

Not every class should be immutable. Immutability makes most sense for classes that represent values, such as strings, dates, currency amounts, colors, and so on.

In mutable classes, it is still a good idea to cleanly separate accessors and mutators, in order to avoid accidental mutation. As a rule of thumb, a method that returns a value should not be a mutator. For example, one would not expect that calling `getBalance` on a `BankAccount` object would change the balance. (You would be pretty upset if your bank charged you a "balance inquiry fee".) If you follow this rule, then all mutators of your class have return type `void`.

Sometimes, this rule is bent a bit, and mutator methods return an informational value. For example, the `ArrayList` class has a `remove` method to remove an object.

```
ArrayList<String> names = ...;
boolean success = names.remove("Romeo");
```

That method returns `true` if the removal was successful; that is, if the list contained the object. Returning this value might be bad design if there was no other way to check whether an object exists in the list. However, there is such a method—the `contains` method. It is acceptable for a mutator to return a value if there is also an accessor that computes it.

The situation is less happy with the `Scanner` class. The `next` method is a mutator that returns a value. (The `next` method really is a mutator. If you call `next` twice in a row, it can return different results, so it must have mutated something inside the `Scanner` object.) Unfortunately, there is no accessor that returns the same value. This sometimes makes it awkward to use a `Scanner`. You must carefully hang on to the value that the `next` method returns because you have no second chance to ask for it. It would have been better if there was another method, say `peek`, that yields the next input without consuming it.

To check the temperature of the water in the bottle, you could take a sip, but that would be the equivalent of a mutator method.

8.2.4 Minimizing Side Effects

A side effect of a method is any externally observable data modification.

A **side effect** of a method is any kind of modification of data that is observable outside the method. Mutator methods have a side effect, namely the modification of the implicit parameter.

There is another kind of side effect that you should avoid. A method should generally not modify its parameter variables. Consider this example:

```
/**
   Computes the total balance of the given accounts.
   @param accounts a list of bank accounts
*/
   public double getTotalBalance(ArrayList<String> accounts)
   {
      double sum = 0;
      while (studentNames.size() > 0)
      {
         BankAccount account = accounts.remove(0); // Not recommended
         sum = sum + account.getBalance();
      }
      return sum;
   }
}
```

This method *removes* all names from the `accounts` parameter variable. After a call

```
double total = getTotalBalance(allAccounts);
```

`allAccounts` is empty! Such a side effect would not be what most programmers expect. It is better if the method visits the elements from the list without removing them.

When designing methods, minimize side effects.

Another example of a side effect is output. Consider again the `printBalance` method that we discussed in Section 8.2.2:

```
public void printBalance() // Not recommended
{
   System.out.println("The balance is now $" + balance);
}
```

This method mutates the `System.out` object, which is not a part of the `BankAccount` object. That is a side effect.

To avoid this side effect, keep most of your classes free from input and output operations, and concentrate input and output in one place, such as the main method of your program.

This taxi has an undesirable side effect, spraying bystanders with muddy water.

3. Why is the `CashRegister` class from Chapter 4 not cohesive?
4. Why does the `Coin` class not depend on the `CashRegister` class?
5. Why is it a good idea to minimize dependencies between classes?
6. Is the `substring` method of the `String` class an accessor or a mutator?
7. Is the `Rectangle` class immutable?
8. If a refers to a bank account, then the call `a.deposit(100)` modifies the bank account object. Is that a side effect?
9. Consider the `Student` class of Chapter 7. Suppose we add a method

```
void read(Scanner in)
{
   while (in.hasNextDouble())
   {
      addScore(in.nextDouble());
   }
}
```

Does this method have a side effect other than mutating the scores?

Practice It Now you can try these exercises at the end of the chapter: R8.4, R8.5, R8.9.

Programming Tip 8.1

Consistency

In this section you learned of two criteria for analyzing the quality of the public interface of a class. You should maximize cohesion and remove unnecessary dependencies. There is another criterion that we would like you to pay attention to—*consistency*. When you have a set of methods, follow a consistent scheme for their names and parameter variables. This is simply a sign of good craftsmanship.

Sadly, you can find any number of inconsistencies in the standard library. Here is an example: To show an input dialog box, you call

```
JOptionPane.showInputDialog(promptString)
```

To show a message dialog box, you call

```
JOptionPane.showMessageDialog(null, messageString)
```

What's the `null` argument? It turns out that the `showMessageDialog` method needs an argument to specify the parent window, or `null` if no parent window is required. But the `showInputDialog` method requires no parent window. Why the inconsistency? There is no reason. It would have been an easy matter to supply a `showMessageDialog` method that exactly mirrors the `showInputDialog` method.

Inconsistencies such as these are not fatal flaws, but they are an annoyance, particularly because they can be so easily avoided.

While it is possible to eat with mismatched silverware, consistency is more pleasant.

Special Topic 8.1

Call by Value and Call by Reference

In Section 8.2.4, we recommended that you don't invoke a mutator method on a parameter variable. In this Special Topic, we discuss a related issue—what happens when you assign a new value to a parameter variable. Consider this method:

```
public class BankAccount
{
   . . .
   /**
      Transfers money from this account and tries to add it to a balance.
      @param amount the amount of money to transfer
      @param otherBalance balance to add the amount to
   */
   public void transfer(double amount, double otherBalance)  (2)
   {
      balance = balance - amount;
      otherBalance = otherBalance + amount;
         // Won't update the argument
   }  (3)
}
```

Now let's see what happens when we call the method:

```
double savingsBalance = 1000;
harrysChecking.transfer(500, savingsBalance);  (1)
System.out.println(savingsBalance);  (4)
```

You might expect that after the call, the `savingsBalance` variable has been incremented to 1500. However, that is not the case. As the method starts, the parameter variable `otherBalance` is set to the same value as `savingsBalance` (see Figure 2). Then the variable is set to a different value. That modification has no effect on `savingsBalance`, because `otherBalance` is a separate variable. When the method terminates, the `otherBalance` variable is removed, and `savingsBalance` isn't increased.

In Java, parameter variables are initialized with the values of the argument expressions. When the method exits, the parameter variables are removed. Computer scientists refer to this call mechanism as "call by value".

> In Java, a method can never change the contents of a variable that is passed to a method.

For that reason, a Java method can never change the contents of a variable that is passed as an argument—the method manipulates a different variable.

Other programming languages such as C++ support a mechanism, called "call by reference", that can change the arguments of a method call. You will sometimes read in Java books

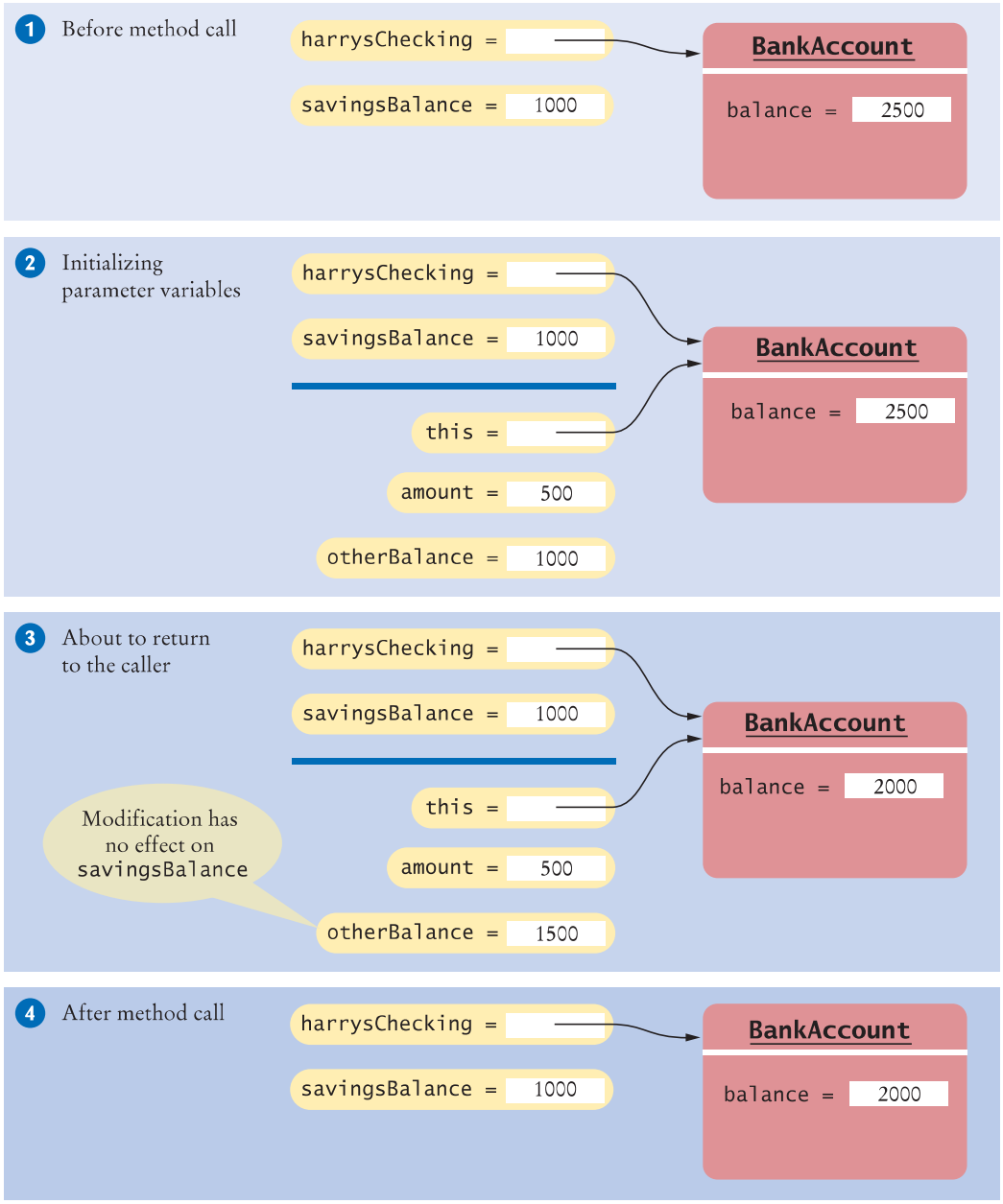


Figure 2 Modifying a Parameter Variable of a Primitive Type Has No Effect on Caller

that "numbers are passed by value, objects are passed by reference". That is technically not quite correct. In Java, objects themselves are never passed as arguments; instead, both numbers and *object references* are passed by value.

The confusion arises because a Java method can mutate an object when it receives an object reference as an argument (see Figure 3).

```
public class BankAccount
{
   . . .
   /**
      Transfers money from this account to another.
      @param amount the amount of money to transfer
      @param otherAccount account to add the amount to
```

```
      */
      public void transfer(double amount, BankAccount otherAccount) 2
      {
         balance = balance - amount;
         otherAccount.deposit(amount);
      } 3
}
```

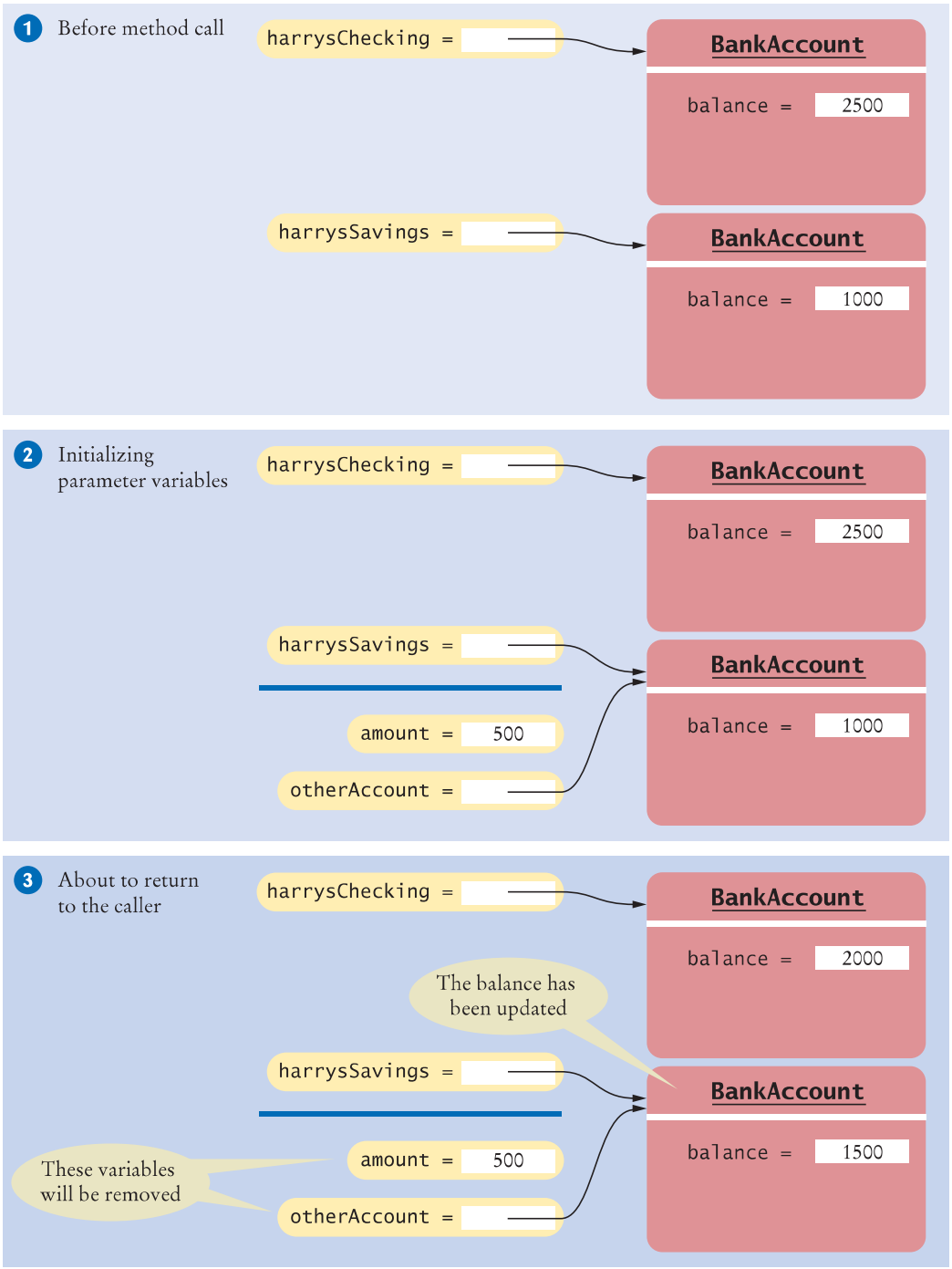


Figure 3 Methods Can Mutate Any Objects to Which They Hold References

Now we pass an object reference to the `transfer` method:

```
BankAccount harrysSavings = new BankAccount(1000);
harrysChecking.transfer(500, harrysSavings); 1
System.out.println(harrysSavings.getBalance());
```

This example works as expected. The parameter variable `otherAccount` contains a *copy* of the object reference `harrysSavings`. You saw in Section 2.8 what is means to make a copy of an object reference—you get another reference to the same object. Through that reference, the method is able to modify the object.

However, a method cannot *replace* an object reference that is passed as an argument. To appreciate this subtle difference, consider this method that tries to set the `otherAccount` parameter variable to a new object:

```
public class BankAccount
{
   . . .
   public void transfer(double amount, BankAccount otherAccount)
   {
      balance = balance - amount;
      double newBalance = otherAccount.balance + amount;
      otherAccount = new BankAccount(newBalance); // Won't work
   }
}
```

In this situation, we are not trying to change the state of the object to which the parameter variable `otherAccount` refers; instead, we are trying to replace the object with a different one (see Figure 4). Now the reference stored in parameter variable `otherAccount` is replaced with a reference to a new account. But if you call the method with

In Java, a method can change the state of an object reference argument, but it cannot replace the object reference with another.

```
harrysChecking.transfer(500, savingsAccount);
```

then that change does not affect the `savingsAccount` variable that is supplied in the call. This example demonstrates that objects are not passed by reference.

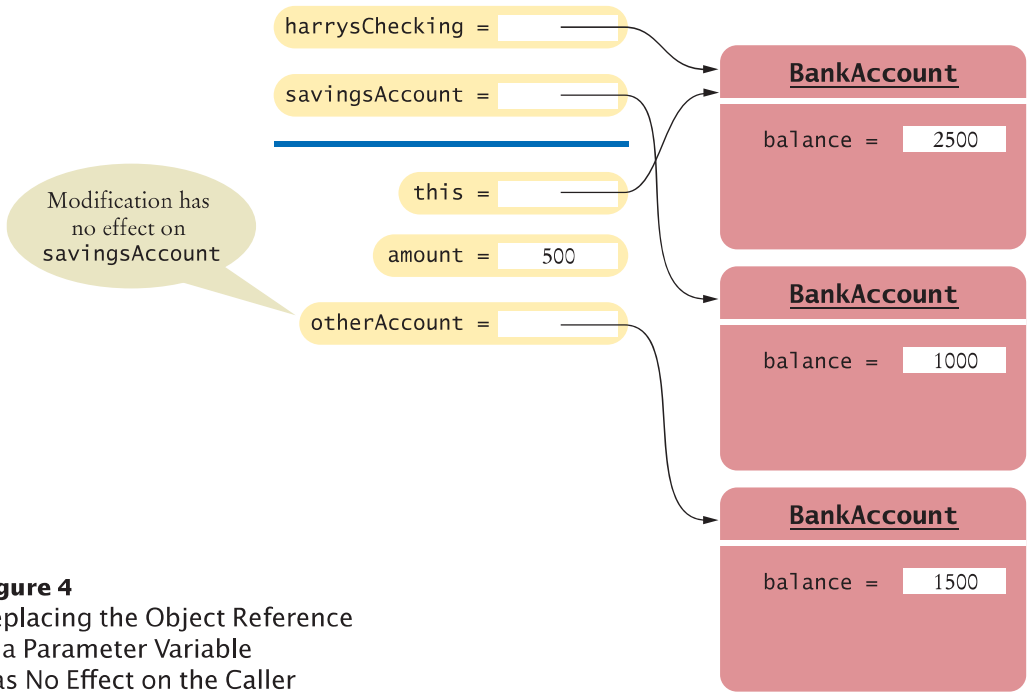


Figure 4
Replacing the Object Reference in a Parameter Variable Has No Effect on the Caller

To summarize:

- A Java method can't change the contents of any variable passed as an argument.
- A Java method can mutate an object when it receives a reference to it as an argument.

8.3 Problem Solving: Patterns for Object Data

When you design a class, you first consider the needs of the programmers who use the class. You provide the methods that the users of your class will call when they manipulate objects. When you implement the class, you need to come up with the instance variables for the class. It is not always obvious how to do this. Fortunately, there is a small set of recurring patterns that you can adapt when you design your own classes. We introduce these patterns in the following sections.

8.3.1 Keeping a Total

An instance variable for the total is updated in methods that increase or decrease the total amount.

Many classes need to keep track of a quantity that can go up or down as certain methods are called. Examples:

- A bank account has a balance that is increased by a deposit, decreased by a withdrawal.
- A cash register has a total that is increased when an item is added to the sale, cleared after the end of the sale.
- A car has gas in the tank, which is increased when fuel is added and decreased when the car drives.

In all of these cases, the implementation strategy is similar. Keep an instance variable that represents the current total. For example, for the cash register:

```
private double purchase;
```

Locate the methods that affect the total. There is usually a method to increase it by a given amount:

```
public void recordPurchase(double amount)
{
   purchase = purchase + amount;
}
```

Depending on the nature of the class, there may be a method that reduces or clears the total. In the case of the cash register, one can provide a `clear` method:

```
public void clear()
{
   purchase = 0;
}
```

There is usually a method that yields the current total. It is easy to implement:

```
public double getAmountDue()
{
   return purchase;
}
```

All classes that manage a total follow the same basic pattern. Find the methods that affect the total and provide the appropriate code for increasing or decreasing it. Find

the methods that report or use the total, and have those methods read the current total.

8.3.2 Counting Events

A counter that counts events is incremented in methods that correspond to the events.

You often need to count how many times certain events occur in the life of an object. For example:

- In a cash register, you may want to know how many items have been added in a sale.
- A bank account charges a fee for each transaction; you need to count them.

Keep a counter, such as

```
private int itemCount;
```

Increment the counter in those methods that correspond to the events that you want to count:

```
public void recordPurchase(double amount)
{
   purchase = purchase + amount;
   itemCount++;
}
```

You may need to clear the counter, for example at the end of a sale or a statement period:

```
public void clear()
{
   purchase = 0;
   itemCount = 0;
}
```

There may or may not be a method that reports the count to the class user. The count may only be used to compute a fee or an average. Find out which methods in your class make use of the count, and read the current value in those methods.

8.3.3 Collecting Values

Some objects collect numbers, strings, or other objects. For example, each multiple-choice question has a number of choices. A cash register may need to store all prices of the current sale.

An object can collect other objects in an array or array list.

Use an array list or an array to store the values. (An array list is usually simpler because you won't need to track the number of values.) For example,

```
public class Question
{
   private ArrayList<String> choices;
   . . .
}
```

A shopping cart object needs to manage a collection of items.

In the constructor, initialize the instance variable to an empty collection:

```
public Question()
{
   choices = new ArrayList<String>();
}
```

You need to supply some mechanism for adding values. It is common to provide a method for appending a value to the collection:

```
public void add(String option)
{
   choices.add(option);
}
```

The user of a `Question` object can call this method multiple times to add the choices.

8.3.4 Managing Properties of an Object

An object property can be accessed with a getter method and changed with a setter method.

A property is a value of an object that an object user can set and retrieve. For example, a `Student` object may have a name and an ID. Provide an instance variable to store the property's value and methods to get and set it.

```
public class Student
{
   private String name;
   . . .
   public String getName() { return name; }
   public void setName(String newName) { name = newName; }
   . . .
}
```

It is common to add error checking to the setter method. For example, we may want to reject a blank name:

```
public void setName(String newName)
{
   if (newName.length() > 0) { name = newName; }
}
```

Some properties should not change after they have been set in the constructor. For example, a student's ID may be fixed (unlike the student's name, which may change). In that case, don't supply a setter method.

```
public class Student
{
   private int id;
   . . .
   public Student(int anId) { id = anId; }
   public String getId() { return id; }
   // No setId method
   . . .
}
```

8.3.5 Modeling Objects with Distinct States

Some objects have behavior that varies depending on what has happened in the past. For example, a `Fish` object may look for food when it is hungry and ignore food after it has eaten. Such an object would need to remember whether it has recently eaten.

If a fish is in a hungry state, its behavior changes.

If your object can have one of several states that affect the behavior, supply an instance variable for the current state.

Supply an instance variable that models the state, together with some constants for the state values:

```
public class Fish
{
   private int hungry;

   public static final int NOT_HUNGRY = 0;
   public static final int SOMEWHAT_HUNGRY = 1;
   public static final int VERY_HUNGRY = 2;
   . . .
}
```

(Alternatively, you can use an enumeration—see Special Topic 5.4.)

Determine which methods change the state. In this example, a fish that has just eaten won't be hungry. But as the fish moves, it will get hungrier:

```
public void eat()
{
   hungry = NOT_HUNGRY;
   . . .
}

public void move()
{
   . . .
   if (hungry < VERY_HUNGRY) { hungry++; }
}
```

Finally, determine where the state affects behavior. A fish that is very hungry will want to look for food first:

```
public void move()
{
   if (hungry == VERY_HUNGRY)
   {
      Look for food.
   }
   . . .
}
```

8.3.6 Describing the Position of an Object

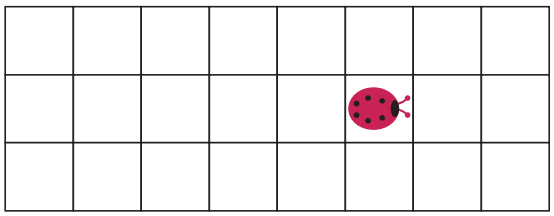
To model a moving object, you need to store and update its position.

Some objects move around during their lifetime, and they remember their current position. For example,

- A train drives along a track and keeps track of the distance from the terminus.
- A simulated bug living on a grid crawls from one grid location to the next, or makes 90 degree turns to the left or right.
- A cannonball is shot into the air, then descends as it is pulled by the gravitational force.

Such objects need to store their position. Depending on the nature of their movement, they may also need to store their orientation or velocity.

A bug in a grid needs to store its row, column, and direction.



If the object moves along a line, you can represent the position as a distance from a fixed point:

```
private double distanceFromTerminus;
```

If the object moves in a grid, remember its current location and direction in the grid:

```
private int row;
private int column;
private int direction; // 0 = North, 1 = East, 2 = South, 3 = West
```

When you model a physical object such as a cannonball, you need to track both the position and the velocity, possibly in two or three dimensions. Here we model a cannonball that is shot upward into the air:

```
private double zPosition;
private double zVelocity;
```

There will be methods that update the position. In the simplest case, you may be told by how much the object moves:

```
public void move(double distanceMoved)
{
   distanceFromTerminus = distanceFromTerminus + distanceMoved;
}
```

If the movement happens in a grid, you need to update the row or column, depending on the current orientation.

```
public void moveOneUnit()
{
   if (direction == NORTH) { row--; }
   else if (direction == EAST) { column++; }
   else if (direction == SOUTH) { row++; }
   else if (direction == WEST) { column--; }
}
```

FULL CODE EXAMPLE

Go to wiley.com/go/javacode to download classes that use these patterns for object data.

Exercise P8.6 shows you how to update the position of a physical object with known velocity.

Whenever you have a moving object, keep in mind that your program will simulate the actual movement in some way. Find out the rules of that simulation, such as movement along a line or in a grid with integer coordinates. Those rules determine how to represent the current position. Then locate the methods that move the object, and update the positions according to the rules of the simulation.

SELF CHECK

10. Suppose we want to count the number of transactions in a bank account in a statement period, and we add a counter to the `BankAccount` class:

```
public class BankAccount
{
   private int transactionCount;
   . . .
```

```
}
```

In which methods does this counter need to be updated?

11. In How To 3.1, the `CashRegister` class does not have a `getTotalPurchase` method. Instead, you have to call `receivePayment` and then `giveChange`. Which recommendation of Section 8.2.4 does this design violate? What is a better alternative?
12. In the example in Section 8.3.3, why is the `add` method required? That is, why can't the user of a `Question` object just call the `add` method of the `ArrayList<String>` class?
13. Suppose we want to enhance the `CashRegister` class in How To 3.1 to track the prices of all purchased items for printing a receipt. Which instance variable should you provide? Which methods should you modify?
14. Consider an `Employee` class with properties for tax ID number and salary. Which of these properties should have only a getter method, and which should have getter and setter methods?
15. Suppose the `setName` method in Section 8.3.4 is changed so that it returns true if the new name is set, false if not. Is this a good idea?
16. Look at the `direction` instance variable in the bug example in Section 8.3.6. This is an example of which pattern?

Practice It Now you can try these exercises at the end of the chapter: E8.21, E8.22, E8.23.

8.4 Static Variables and Methods

A static variable belongs to the class, not to any object of the class.

Sometimes, a value properly belongs to a class, not to any object of the class. You use a **static variable** for this purpose. Here is a typical example: We want to assign bank account numbers sequentially. That is, we want the bank account constructor to construct the first account with number 1001, the next with number 1002, and so on. To solve this problem, we need to have a single value of `lastAssignedNumber` that is a property of the *class*, not any object of the class. Such a variable is called a static variable because you declare it using the `static` reserved word.

The reserved word `static` *is a holdover from the C++ language. Its use in Java has no relationship to the normal use of the term.*

```
public class BankAccount
{
   private double balance;
   private int accountNumber;
   private static int lastAssignedNumber = 1000;

   public BankAccount()
   {
      lastAssignedNumber++;
      accountNumber = lastAssignedNumber;
   }
   . . .
}
```

Every BankAccount object has its own balance and accountNumber instance variables, but all objects share a single copy of the lastAssignedNumber variable (see Figure 5). That variable is stored in a separate location, outside any BankAccount objects.

Like instance variables, static variables should always be declared as private to ensure that methods of other classes do not change their values. However, static *constants* may be either private or public.

For example, the BankAccount class can define a public constant value, such as

```
public class BankAccount
{
   public static final double OVERDRAFT_FEE = 29.95;
   . . .
}
```

Methods from any class can refer to such a constant as BankAccount.OVERDRAFT_FEE.

A static method is not invoked on an object.

Sometimes a class defines methods that are not invoked on an object. Such a method is called a **static method**. A typical example of a static method is the sqrt method in the Math class. Because numbers aren't objects, you can't invoke methods on them. For example, if x is a number, then the call x.sqrt() is not legal in Java. Therefore, the Math class provides a static method that is invoked as Math.sqrt(x). No object of the Math class is constructed. The Math qualifier simply tells the compiler where to find the sqrt method.

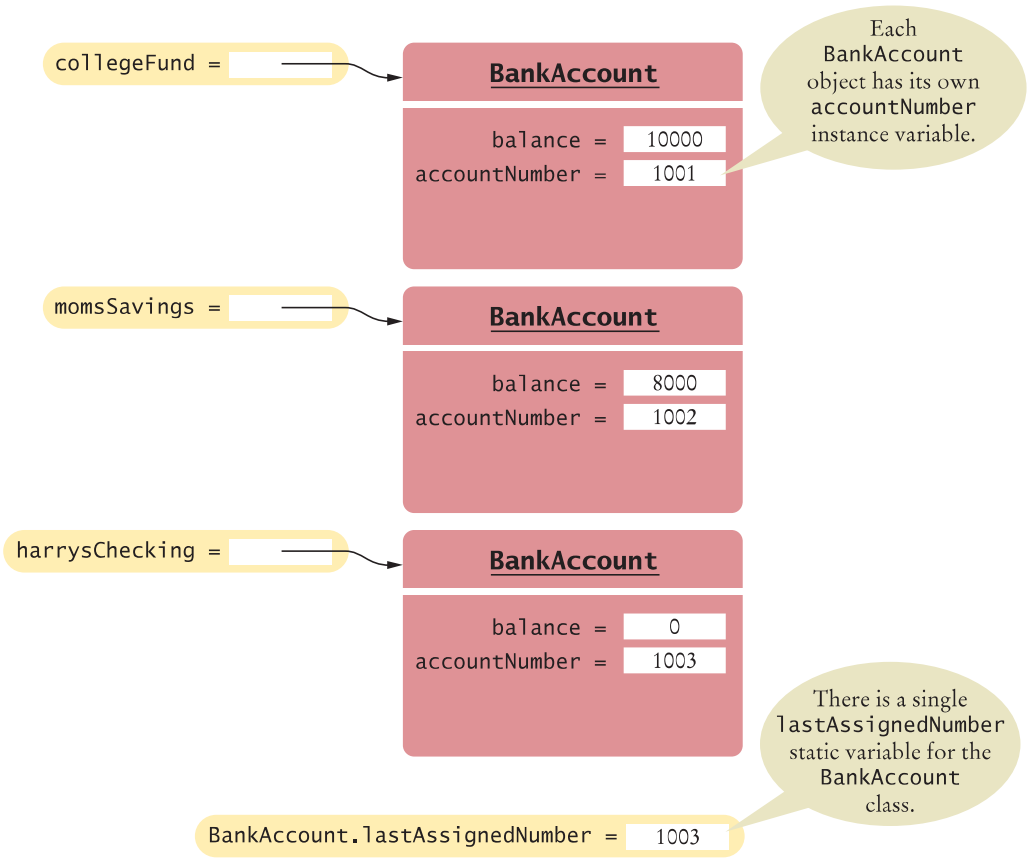


Figure 5 A Static Variable and Instance Variables

You can define your own static methods for use in other classes. Here is an example:

```
public class Financial
{
   /**
      Computes a percentage of an amount.
      @param percentage the percentage to apply
      @param amount the amount to which the percentage is applied
      @return the requested percentage of the amount
   */
   public static double percentOf(double percentage, double amount)
   {
      return (percentage / 100) * amount;
   }
}
```

FULL CODE EXAMPLE

Go to wiley.com/go/javacode to download a program with static methods and variables.

When calling this method, supply the name of the class containing it:

```
double tax = Financial.percentOf(taxRate, total);
```

In object-oriented programming, static methods are not very common. Nevertheless, the `main` method is always static. When the program starts, there aren't any objects. Therefore, the first method of a program must be a static method.

SELF CHECK

17. Name two static variables of the `System` class.
18. Name a static constant of the `Math` class.
19. The following method computes the average of an array of numbers:

    ```
    public static double average(double[] values)
    ```

    Why should it not be defined as an instance method?
20. Harry tells you that he has found a great way to avoid those pesky objects: Put all code into a single class and declare all methods and variables `static`. Then `main` can call the other static methods, and all of them can access the static variables. Will Harry's plan work? Is it a good idea?

Practice It Now you can try these exercises at the end of the chapter: R8.22, E8.5, E8.6.

Programming Tip 8.2

Minimize the Use of Static Methods

It is possible to solve programming problems by using classes with only static methods. In fact, before object-oriented programming was invented, that approach was quite common. However, it usually leads to a design that is not object-oriented and makes it hard to evolve a program.

Consider the task of How To 7.1. A program reads scores for a student and prints the final score, which is obtained by dropping the lowest one. We solved the problem by implementing a `Student` class that stores student scores. Of course, we could have simply written a program with a few static methods:

```
public class ScoreAnalyzer
{
   public static double[] readInputs() { . . . }
   public static double sum(double[] values) { . . . }
   public static double minimum(double[] values) { . . . }
```

```
   public static double finalScore(double[] values)
   {
      if (values.length == 0) { return 0; }
      else if (values.length == 1) { return values[0]; }
      else { return sum(values) - minimum(values); }
   }

   public static void main(String[] args)
   {
      System.out.println(finalScore(readInputs()));
   }
}
```

That solution is fine if one's sole objective is to solve a simple homework problem. But suppose you need to modify the program so that it deals with multiple students. An object-oriented program can evolve the `Student` class to store grades for many students. In contrast, adding more functionality to static methods gets messy quickly (see Exercise E8.7).

Common Error 8.1

Trying to Access Instance Variables in Static Methods

A static method does not operate on an object. In other words, it has no implicit parameter, and you cannot directly access any instance variables. For example, the following code is wrong:

```
public class SavingsAccount
{
   private double balance;
   private double interestRate;

   public static double interest(double amount)
   {
      return (interestRate / 100) * amount;
      // ERROR: Static method accesses instance variable
   }
}
```

Because different savings accounts can have different interest rates, the `interest` method should not be a static method.

Special Topic 8.2

Static Imports

Starting with Java version 5.0, there is a variant of the `import` directive that lets you use static methods and variables without class prefixes. For example,

```
import static java.lang.System.*;
import static java.lang.Math.*;

public class RootTester
{
   public static void main(String[] args)
   {
      double r = sqrt(PI); // Instead of Math.sqrt(Math.PI)
      out.println(r);      // Instead of System.out
   }
```

```
}
```

Static imports can make programs easier to read, particularly if they use many mathematical functions.

Special Topic 8.3

Alternative Forms of Instance and Static Variable Initialization

As you have seen, instance variables are initialized with a default value (`0`, `false`, or `null`, depending on their type). You can then set them to any desired value in a constructor, and that is the style that we prefer in this book.

However, there are two other mechanisms to specify an initial value. Just as with local variables, you can specify initialization values for instance variables. For example,

```
public class Coin
{
   private double value = 1;
   private String name = "Dollar";
   . . .
}
```

These default values are used for *every* object that is being constructed.

There is also another, much less common, syntax. You can place one or more *initialization blocks* inside the class declaration. All statements in that block are executed whenever an object is being constructed. Here is an example:

```
public class Coin
{
   private double value;
   private String name;
   {
      value = 1;
      name = "Dollar";
   }
   . . .
}
```

For static variables, you use a static initialization block:

```
public class BankAccount
{
   private static int lastAssignedNumber;
   static
   {
       lastAssignedNumber = 1000;
   }
   . . .
}
```

All statements in the static initialization block are executed once when the class is loaded. Initialization blocks are rarely used in practice.

When an object is constructed, the initializers and initialization blocks are executed in the order in which they appear. Then the code in the constructor is executed. Because the rules for the alternative initialization mechanisms are somewhat complex, we recommend that you simply use constructors to do the job of construction.

8.5 Packages

A package is a set of related classes.

A Java program consists of a collection of classes. So far, most of your programs have consisted of a small number of classes. As programs get larger, however, simply distributing the classes over multiple files isn't enough. An additional structuring mechanism is needed.

In Java, packages provide this structuring mechanism. A Java **package** is a set of related classes. For example, the Java library consists of several hundred packages, some of which are listed in Table 1.

Table 1 Important Packages in the Java Library

Package	Purpose	Sample Class
`java.lang`	Language support	`Math`
`java.util`	Utilities	`Random`
`java.io`	Input and output	`PrintStream`
`java.awt`	Abstract Windowing Toolkit	`Color`
`java.applet`	Applets	`Applet`
`java.net`	Networking	`Socket`
`java.sql`	Database access through Structured Query Language	`ResultSet`
`javax.swing`	Swing user interface	`JButton`
`org.w3c.dom`	Document Object Model for XML documents	`Document`

8.5.1 Organizing Related Classes into Packages

To put one of your classes in a package, you must place a line

```
package packageName;
```

as the first instruction in the source file containing the class. A package name consists of one or more identifiers separated by periods. (See Section 8.5.3 for tips on constructing package names.)

For example, let's put the `Financial` class introduced in this chapter into a package named `com.horstmann.bigjava`. The `Financial.java` file must start as follows:

```
package com.horstmann.bigjava;
public class Financial
{
   . . .
}
```

In addition to the named packages (such as `java.util` or `com.horstmann.bigjava`), there is a special package, called the *default package*, which has no name. If you did not

In Java, related classes are grouped into packages.

include any `package` statement at the top of your source file, its classes are placed in the default package.

8.5.2 Importing Packages

If you want to use a class from a package, you can refer to it by its full name (package name plus class name). For example, `java.util.Scanner` refers to the `Scanner` class in the `java.util` package:

```
java.util.Scanner in = new java.util.Scanner(System.in);
```

The `import` directive lets you refer to a class of a package by its class name, without the package prefix.

Naturally, that is somewhat inconvenient. For that reason, you usually import a name with an `import` statement:

```
import java.util.Scanner;
```

Then you can refer to the class as `Scanner` without the package prefix.

You can import *all classes* of a package with an `import` statement that ends in `.*`. For example, you can use the statement

```
import java.util.*;
```

to import all classes from the `java.util` package. That statement lets you refer to classes like `Scanner` or `Random` without a `java.util` prefix.

However, you never need to import the classes in the `java.lang` package explicitly. That is the package containing the most basic Java classes, such as `Math` and `Object`. These classes are always available to you. In effect, an automatic `import java.lang.*;` statement has been placed into every source file.

Finally, you don't need to import other classes in the same package. For example, when you implement the class `homework1.Tester`, you don't need to import the class `homework1.Bank`. The compiler will find the `Bank` class without an `import` statement because it is located in the same package, `homework1`.

8.5.3 Package Names

Placing related classes into a package is clearly a convenient mechanism to organize classes. However, there is a more important reason for packages: to avoid **name clashes**. In a large project, it is inevitable that two people will come up with the same name for the same concept. This even happens in the standard Java class library (which has now grown to thousands of classes). There is a class `Timer` in the `java.util`

Syntax 8.1 Package Specification

Syntax `package` *packageName*;

```
package com.horstmann.bigjava;
```

The classes in this file belong to this package.

A good choice for a package name is a domain name in reverse.

package and another class called `Timer` in the `javax.swing` package. You can still tell the Java compiler exactly which `Timer` class you need, simply by referring to them as `java.util.Timer` and `javax.swing.Timer`.

Of course, for the package-naming convention to work, there must be some way to ensure that package names are unique. It wouldn't be good if the car maker BMW placed all its Java code into the package `bmw`, and some other programmer (perhaps Britney M. Walters) had the same bright idea. To avoid this problem, the inventors of Java recommend that you use a package-naming scheme that takes advantage of the uniqueness of Internet domain names.

Use a domain name in reverse to construct an unambiguous package name.

For example, I have a domain name `horstmann.com`, and there is nobody else on the planet with the same domain name. (I was lucky that the domain name `horstmann.com` had not been taken by anyone else when I applied. If your name is Walters, you will sadly find that someone else beat you to `walters.com`.) To get a package name, turn the domain name around to produce a package name prefix, such as `com.horstmann`.

If you don't have your own domain name, you can still create a package name that has a high probability of being unique by writing your e-mail address backwards. For example, if Britney Walters has an e-mail address `walters@cs.sjsu.edu`, then she can use a package name `edu.sjsu.cs.walters` for her own classes.

Some instructors will want you to place each of your assignments into a separate package, such as `homework1`, `homework2`, and so on. The reason is again to avoid name collision. You can have two classes, `homework1.Bank` and `homework2.Bank`, with slightly different properties.

8.5.4 Packages and Source Files

The path of a class file must match its package name.

A source file must be located in a subdirectory that matches the package name. The parts of the name between periods represent successively nested directories. For example, the source files for classes in the package `com.horstmann.bigjava` would be placed in a subdirectory `com/horstmann/bigjava`. You place the subdirectory inside the *base directory* holding your program's files. For example, if you do your homework assignment in a directory `/home/britney/hw8/problem1`, then you can place the class files for the `com.horstmann.bigjava` package into the directory `/home/britney/hw8/problem1/com/horstmann/bigjava`, as shown in Figure 6. (Here, we are using UNIX-style file names. Under Windows, you might use `c:\Users\Britney\hw8\problem1\com\horstmann\bigjava`.)

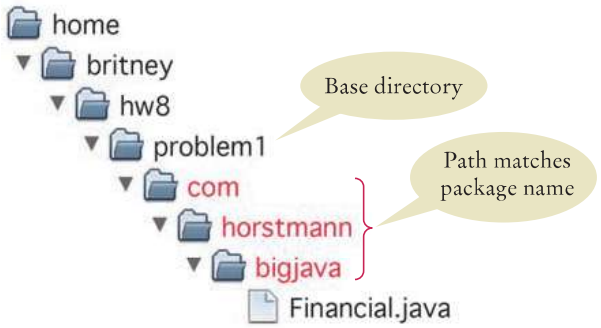


Figure 6 Base Directories and Subdirectories for Packages

21. Which of the following are packages?

a. `java`

b. `java.lang`

c. `java.util`

d. `java.lang.Math`

22. Is a Java program without `import` statements limited to using the default and `java.lang` packages?

23. Suppose your homework assignments are located in the directory `/home/me/cs101` (`c:\Users\Me\cs101` on Windows). Your instructor tells you to place your homework into packages. In which directory do you place the class `hw1.problem1.TicTacToeTester`?

Practice It Now you can try these exercises at the end of the chapter: R8.25, E8.15, E8.16.

Common Error 8.2

Confusing Dots

In Java, the dot symbol (`.`) is used as a separator in the following situations:

- Between package names (`java.util`)
- Between package and class names (`homework1.Bank`)
- Between class and inner class names (`Ellipse2D.Double`)
- Between class and instance variable names (`Math.PI`)
- Between objects and methods (`account.getBalance()`)

When you see a long chain of dot-separated names, it can be a challenge to find out which part is the package name, which part is the class name, which part is an instance variable name, and which part is a method name. Consider

```
java.lang.System.out.println(x);
```

Because `println` is followed by an opening parenthesis, it must be a method name. Therefore, `out` must be either an object or a class with a static `println` method. (Of course, we know that `out` is an object reference of type `PrintStream`.) Again, it is not at all clear, without context, whether `System` is another object, with a public variable `out`, or a class with a `static` variable.

Judging from the number of pages that the Java language specification devotes to this issue, even the compiler has trouble interpreting these dot-separated sequences of strings.

To avoid problems, it is helpful to adopt a strict coding style. If class names always start with an uppercase letter, and variable, method, and package names always start with a lowercase letter, then confusion can be avoided.

Special Topic 8.4

Package Access

If a class, instance variable, or method has no `public` or `private` modifier, then all methods of classes in the same package can access the feature. For example, if a class is declared as `public`, then all other classes in all packages can use it. But if a class is declared without an access modifier, then only the other classes in the *same* package can use it. Package access is a reasonable default for classes, but it is extremely unfortunate for instance variables.

An instance variable or method that is not declared as `public` or `private` can be accessed by all classes in the same package, which is usually not desirable.

It is a common error to *forget* the reserved word `private`, thereby opening up a potential security hole. For example, at the time of this writing, the `Window` class in the `java.awt` package contained the following declaration:

```
public class Window extends Container
{
   String warningString;
   . . .
}
```

There actually was no good reason to grant package access to the `warningString` instance variable—no other class accesses it.

Package access for instance variables is rarely useful and always a potential security risk. Most instance variables are given package access by accident because the programmer simply forgot the `private` reserved word. It is a good idea to get into the habit of scanning your instance variable declarations for missing `private` modifiers.

HOW TO 8.1

Programming with Packages

This How To explains in detail how to place your programs into packages.

Problem Statement Place each homework assignment into a separate package. That way, you can have classes with the same name but different implementations in separate packages (such as `homework1.problem1.Bank` and `homework1.problem2.Bank`).

Step 1 Come up with a package name.

Your instructor may give you a package name to use, such as `homework1.problem2`. Or, perhaps you want to use a package name that is unique to you. Start with your e-mail address, written backwards. For example, `walters@cs.sjsu.edu` becomes `edu.sjsu.cs.walters`. Then add a subpackage that describes your project, such as `edu.sjsu.cs.walters.cs1project`.

Step 2 Pick a *base directory*.

The base directory is the directory that contains the directories for your various packages, for example, `/home/britney` or `c:\Users\Britney`.

Step 3 Make a subdirectory from the base directory that matches your package name.

The subdirectory must be contained in your base directory. Each segment must match a segment of the package name. For example,

```
mkdir -p /home/britney/homework1/problem2 (in UNIX)
```

or

```
mkdir /s c:\Users\Britney\homework1\problem2 (in Windows)
```

Step 4 Place your source files into the package subdirectory.

For example, if your homework consists of the files `Tester.java` and `Bank.java`, then you place them into

```
/home/britney/homework1/problem2/Tester.java
/home/britney/homework1/problem2/Bank.java
```

or

```
c:\Users\Britney\homework1\problem2\Tester.java
c:\Users\Britney\homework1\problem2\Bank.java
```

Step 5 Use the `package` statement in each source file.

The first noncomment line of each file must be a package statement that lists the name of the package, such as

```
package homework1.problem2;
```

Step 6 Compile your source files from the *base directory*.

Change to the base directory (from Step 2) to compile your files. For example,

```
cd /home/britney
javac homework1/problem2/Tester.java
```

or

```
c:
cd \Users\Britney
javac homework1\problem2\Tester.java
```

Note that the Java compiler needs the *source file name and not the class name.* That is, you need to supply file separators (`/` on UNIX, `\` on Windows) and a file extension (`.java`).

Step 7 Run your program from the *base directory*.

Unlike the Java compiler, the Java interpreter needs the *class name (and not a file name) of the class containing the* `main` *method*. That is, use periods as package separators, and don't use a file extension. For example,

```
cd /home/britney
java homework1.problem2.Tester
```

or

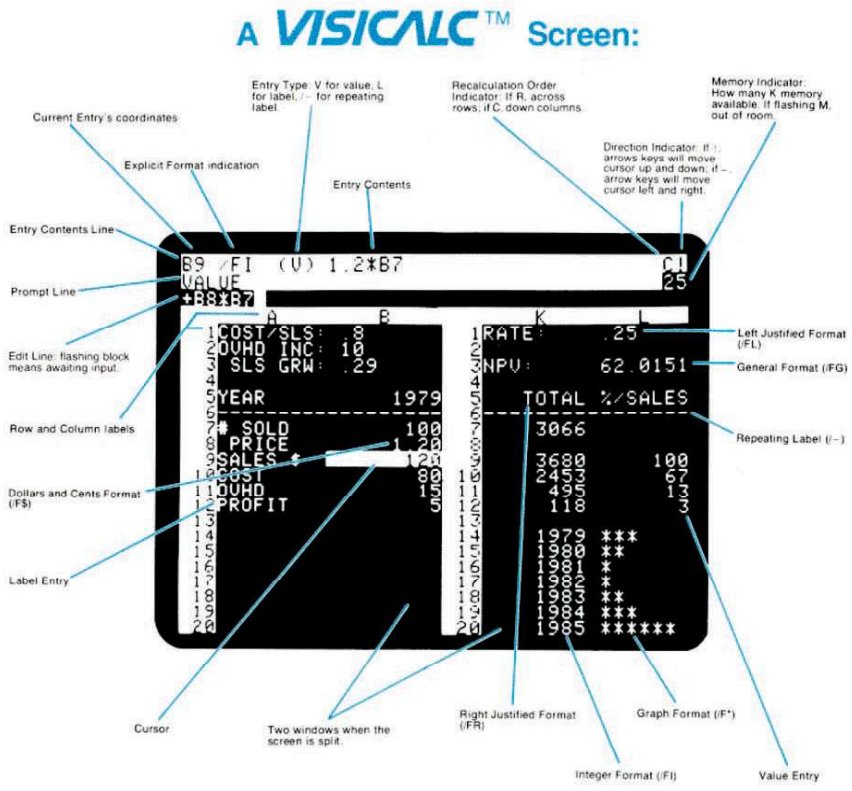
```
c:
cd \Users\Britney
java homework1.problem2.Tester
```

Computing & Society 8.1 Personal Computing

In 1971, Marcian E. "Ted" Hoff, an engineer at Intel Corporation, was working on a chip for a manufacturer of electronic calculators. He realized that it would be a better idea to develop a *general-purpose* chip that could be *programmed* to interface with the keys and display of a calculator, rather than to do yet another custom design. Thus, the *microprocessor* was born. At the time, its primary application was as a controller for calculators, washing machines, and the like. It took years for the computer industry to notice that a genuine central processing unit was now available as a single chip.

Hobbyists were the first to catch on. In 1974 the first computer *kit,* the Altair 8800, was available from MITS Electronics for about $350. The kit consisted of the microprocessor, a circuit board, a very small amount of memory, toggle switches, and a row of display lights. Purchasers had to solder and assemble it, then program it in machine language through the toggle switches. It was not a big hit.

The first big hit was the Apple II. It was a real computer with a keyboard, a monitor, and a floppy disk drive. When it was first released, users had a $3,000 machine that could play Space Invaders, run a primitive bookkeeping program, or let users program it in BASIC. The original Apple II did not even support lowercase letters, making it worthless for word processing. The breakthrough came in 1979 with a new spreadsheet program, VisiCalc. In a spreadsheet, you enter financial data and their relationships into a grid of rows and columns (see the figure). Then you modify some of the data and watch in real time how the others change. For example, you can see how changing the mix of widgets in a manufacturing plant might affect estimated costs and profits. Corporate managers snapped up VisiCalc and the computer that was needed to run it. For them, the computer was a spreadsheet machine. More importantly, it was a personal device. The managers were free to do the calculations that they wanted to do, not just the ones that the "high priests" in the data center provided.



The Visicalc Spreadsheet Running on an Apple II

Personal computers have been with us ever since, and countless users have tinkered with their hardware and software, sometimes establishing highly successful companies or creating free software for millions of users. This "freedom to tinker" is an important part of personal computing. On a personal device, you should be able to install the software that you want to install to make you more productive or creative, even if that's not the same software that most people use. You should be able to add peripheral equipment of your choice. For the first thirty years of personal computing, this freedom was largely taken for granted.

We are now entering an era where smartphones, tablets, and smart TV sets are replacing functions that were traditionally fulfilled by personal computers. While it is amazing to carry more computing power in your cell phone than in the best personal computers of the 1990s, it is disturbing that we lose a degree of personal control. With some phone or tablet brands, you can only install those applications that the manufacturer publishes on the "app store". For example, Apple does not allow children to learn the Scratch language on the iPad. You'd think it would be in Apple's interest to encourage the next generation to be enthusiastic about programming, but they have a general policy of denying programmability on "their" devices, in order to thwart competitive environments such as Flash or Java.

When you select a device for making phone calls or watching movies, it is worth asking who is in control. Are you purchasing a personal device that you can use in any way you choose, or are you being tethered to a flow of data that is controlled by somebody else?

8.6 Unit Test Frameworks

Unit test frameworks simplify the task of writing classes that contain many test cases.

Up to now, we have used a very simple approach to testing. We provided tester classes whose `main` method computes values and prints actual and expected values. However, that approach has limitations. The `main` method gets messy if it contains many tests. And if an exception occurs during one of the tests, the remaining tests are not executed.

Unit testing frameworks were designed to quickly execute and evaluate test suites and to make it easy to incrementally add test cases. One of the most popular testing frameworks is JUnit. It is freely available at `http://junit.org`, and it is also built into a number of development environments, including BlueJ and Eclipse. Here we describe JUnit 4, the most current version of the library as this book is written.

When you use JUnit, you design a companion test class for each class that you develop. You provide a method for each test case that you want to have executed. You use "annotations" to mark the test methods. An annotation is an advanced Java feature that places a marker into the code that is interpreted by another tool. In the case of JUnit, the `@Test` annotation is used to mark test methods.

In each test case, you make some computations and then compute some condition that you believe to be true. You then pass the result to a method that communicates a test result to the framework, most commonly the `assertEquals` method. The `assertEquals` method takes as arguments the expected and actual values and, for floating-point numbers, a tolerance value.

It is also customary (but not required) that the name of the test class ends in `Test`, such as `CashRegisterTest`. Here is a typical example:

```
import org.junit.Test;
import org.junit.Assert;

public class CashRegisterTest
{
   @Test public void twoPurchases()
   {
      CashRegister register = new CashRegister();
      register.recordPurchase(0.75);
      register.recordPurchase(1.50);
      register.receivePayment(2, 0, 5, 0, 0);
      double expected = 0.25;
      Assert.assertEquals(expected, register.giveChange(), EPSILON);
   }
   // More test cases
   . . .
}
```

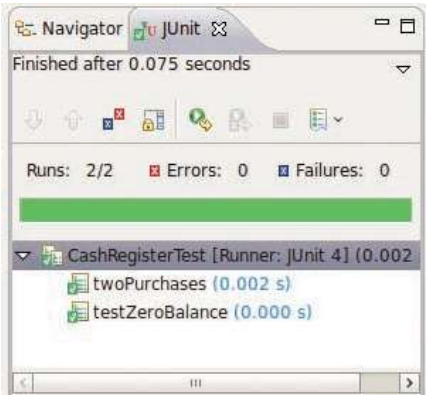
If all test cases pass, the JUnit tool shows a green bar (see Figure 7). If any of the test cases fail, the JUnit tool shows a red bar and an error message.

Your test class can also have other methods (whose names should not be annotated with `@Test`). These methods typically carry out steps that you want to share among test methods.

The JUnit philosophy is to run all tests whenever you change your code.

The JUnit philosophy is simple. Whenever you implement a class, also make a companion test class. You design the tests as you design the program, one test method at a time. The test cases just keep accumulating in the test class. Whenever you have detected an actual failure, add a test case that flushes it out, so that you can be sure

Figure 7
Unit Testing with JUnit



that you won't introduce that particular bug again. Whenever you modify your class, simply run the tests again.

If all tests pass, the user interface shows a green bar and you can relax. Otherwise, there is a red bar, but that's also good. It is much easier to fix a bug in isolation than inside a complex program.

SELF CHECK

24. Provide a JUnit test class with one test case for the `Earthquake` class in Chapter 5.

25. What is the significance of the `EPSILON` argument in the `assertEquals` method?

Practice It Now you can try these exercises at the end of the chapter: R8.27, E8.17, E8.18.

CHAPTER SUMMARY

Find classes that are appropriate for solving a programming problem.

- A class should represent a single concept from a problem domain, such as business, science, or mathematics.

Design methods that are cohesive, consistent, and minimize side effects.

- The public interface of a class is cohesive if all of its features are related to the concept that the class represents.
- A class depends on another class if its methods use that class in any way.
- An immutable class has no mutator methods.
- References to objects of an immutable class can be safely shared.

- A side effect of a method is any externally observable data modification.
- When designing methods, minimize side effects.
- In Java, a method can never change the contents of a variable that is passed to a method.
- In Java, a method can change the state of an object reference argument, but it cannot replace the object reference with another.

Use patterns to design the data representation of an object.

- An instance variable for the total is updated in methods that increase or decrease the total amount.
- A counter that counts events is incremented in methods that correspond to the events.
- An object can collect other objects in an array or array list.
- An object property can be accessed" with a getter method and changed with a setter method.
- If your object can have one of several states that affect the behavior, supply an instance variable for the current state.
- To model a moving object, you need to store and update its position.

Understand the behavior of static variables and static methods.

- A static variable belongs to the class, not to any object of the class.
- A static method is not invoked on an object.

Use packages to organize sets of related classes.

- A package is a set of related classes.
- The `import` directive lets you refer to a class of a package by its class name, without the package prefix.
- Use a domain name in reverse to construct an unambiguous package name.
- The path of a class file must match its package name.
- An instance variable or method that is not declared as `public` or `private` can be accessed by all classes in the same package, which is usually not desirable.

Use JUnit for writing unit tests.

- Unit test frameworks simplify the task of writing classes that contain many test cases.
- The JUnit philosophy is to run all tests whenever you change your code.

REVIEW QUESTIONS

■■ **R8.1** Your task is to write a program that simulates a vending machine. Users select a product and provide payment. If the payment is sufficient to cover the purchase price of the product, the product is dispensed and change is given. Otherwise, the payment is returned to the user. Name an appropriate class for implementing this program. Name two classes that would not be appropriate and explain why.

■■ **R8.2** Your task is to write a program that reads a customer's name and address, followed by a sequence of purchased items and their prices, and prints an invoice.

Discuss which of the following would be good classes for implementing this program:

a. `Invoice`

b. `InvoicePrinter`

c. `PrintInvoice`

d. `InvoiceProgram`

■■■ **R8.3** Your task is to write a program that computes paychecks. Employees are paid an hourly rate for each hour worked; however, if they worked more than 40 hours per week, they are paid at 150 percent of the regular rate for those overtime hours. Name an actor class that would be appropriate for implementing this program. Then name a class that isn't an actor class that would be an appropriate alternative. How does the choice between these alternatives affect the program structure?

■■ **R8.4** Look at the public interface of the `java.lang.System` class and discuss whether or not it is cohesive.

■■ **R8.5** Suppose an `Invoice` object contains descriptions of the products ordered, and the billing and shipping addresses of the customer. Draw a UML diagram showing the dependencies between the classes `Invoice`, `Address`, `Customer`, and `Product`.

■■ **R8.6** Suppose a vending machine contains products, and users insert coins into the vending machine to purchase products. Draw a UML diagram showing the dependencies between the classes `VendingMachine`, `Coin`, and `Product`.

■■ **R8.7** On which classes does the class `Integer` in the standard library depend?

■■ **R8.8** On which classes does the class `Rectangle` in the standard library depend?

■ **R8.9** Classify the methods of the class `Scanner` that are used in this book as accessors and mutators.

■ **R8.10** Classify the methods of the class `Rectangle` as accessors and mutators.

■ **R8.11** Is the `Resistor` class in Exercise P8.8 a mutable or immutable class? Why?

■ **R8.12** Which of the following classes are immutable?

a. `Rectangle`

b. `String`

c. `Random`

■ **R8.13** Which of the following classes are immutable?

a. `PrintStream`

b. `Date`

c. `Integer`

R8.14 Consider a method

```
public class DataSet
{
   /**
      Reads all numbers from a scanner and adds them to this data set.
      @param in a Scanner
   */
   public void read(Scanner in) { . . . }
   . . .
```

```
}
```

Describe the side effects of the `read` method. Which of them are not recommended, according to Section 8.2.4? Which redesign eliminates the unwanted side effect? What is the effect of the redesign on coupling?

■■ **R8.15** What side effect, if any, do the following three methods have?

```
public class Coin
{
   . . .
   public void print()
   {
      System.out.println(name + " " + value);
   }

   public void print(PrintStream stream)
   {
      stream.println(name + " " + value);
   }

   public String toString()
   {
      return name + " " + value;
   }
}
```

■■■ **R8.16** Ideally, a method should have no side effects. Can you write a program in which no method has a side effect? Would such a program be useful?

■■ **R8.17** Consider the following method that is intended to swap the values of two integers:

```
public static void falseSwap(int a, int b)
{
   int temp = a;
   a = b;
   b = temp;
}

public static void main(String[] args)
{
   int x = 3;
   int y = 4;
   falseSwap(x, y);
   System.out.println(x + " " + y);
}
```

Why doesn't the method swap the contents of `x` and `y`?

■■■ **R8.18** How can you write a method that swaps two floating-point numbers? *Hint:* `java.awt.Point`.

■■ **R8.19** Draw a memory diagram that shows why the following method can't swap two `BankAccount` objects:

```
public static void falseSwap(BankAccount a, BankAccount b)
{
   BankAccount temp = a;
   a = b;
   b = temp;
}
```

■ **R8.20** Consider an enhancement of the `Die` class of Chapter 6 with a static variable

```
public class Die
{
   private int sides;
   private static Random generator = new Random();
   public Die(int s) { . . . }
   public int cast() { . . . }
}
```

Draw a memory diagram that shows three dice:

```
Die d4 = new Die(4);
Die d6 = new Die(6);
Die d8 = new Die(8);
```

Be sure to indicate the values of the `sides` and `generator` variables.

■ **R8.21** Try compiling the following program. Explain the error message that you get.

```
public class Print13
{
   public void print(int x)
   {
      System.out.println(x);
   }

   public static void main(String[] args)
   {
      int n = 13;
      print(n);
   }
}
```

■ **R8.22** Look at the methods in the `Integer` class. Which are static? Why?

■■ **R8.23** Look at the methods in the `String` class (but ignore the ones that take an argument of type `char[]`). Which are static? Why?

■■ **R8.24** The `in` and `out` variables of the `System` class are public static variables of the `System` class. Is that good design? If not, how could you improve on it?

■■ **R8.25** Every Java program can be rewritten to avoid `import` statements. Explain how, and rewrite `RectangleComponent.java` from Section 2.9.3 to avoid `import` statements.

■ **R8.26** What is the default package? Have you used it before this chapter in your programming?

■■ **Testing R8.27** What does JUnit do when a test method throws an exception? Try it out and report your findings.

PRACTICE EXERCISES

■■ **E8.1** Implement the `Coin` class described in Section 8.2. Modify the `CashRegister` class so that coins can be added to the cash register, by supplying a method

```
void receivePayment(int coinCount, Coin coinType)
```

The caller needs to invoke this method multiple times, once for each type of coin that is present in the payment.