CHAPTER 7

ARRAYS AND ARRAY LISTS

© traveler1116/iStockphoto.

CHAPTER GOALS

To collect elements using arrays and array lists

To use the enhanced for loop for traversing arrays and array lists

To learn common algorithms for processing arrays and array lists

To work with two-dimensional arrays

To understand the concept of regression testing

CHAPTER CONTENTS

In many programs, you need to collect large numbers of values. In Java, you use the array and array list constructs for this purpose. Arrays have a more concise syntax, whereas array lists can automatically grow to any desired size. In this chapter, you will learn about arrays, array lists, and common algorithms for processing them.

7.1 Arrays

We start this chapter by introducing the array data type. Arrays are the fundamental mechanism in Java for collecting multiple values. In the following sections, you will learn how to declare arrays and how to access array elements.

7.1.1 Declaring and Using Arrays

Suppose you write a program that reads a sequence of values and prints out the sequence, marking the largest value, like this:

```
32
54
67.5
29
35
80
115 <= largest value
44.5
100
65
```

You do not know which value to mark as the largest one until you have seen them all. After all, the last value might be the largest one. Therefore, the program must first store all values before it can print them.

Could you simply store each value in a separate variable? If you know that there are ten values, then you could store the values in ten variables value1, value2, value3, ..., value10. However, such a sequence of variables is not very practical to use. You would have to write quite a bit of code ten times, once for each of the variables. In Java, an **array** is a much better choice for storing a sequence of values of the same type.

An array collects a sequence of values of the same type.

Here we create an array that can hold ten values of type double:

```
new double[10]
```

The number of elements (here, 10) is called the *length* of the array.

The new operator constructs the array. You will want to store the array in a variable so that you can access it later.

The type of an array variable is the type of the element to be stored, followed by []. In this example, the type is double[], because the element type is double.

Here is the declaration of an array variable of type double[] (see Figure 1):

```
double[] values; ❶
```

When you declare an array variable, it is not yet initialized. You need to initialize the variable with the array:

```
double[] values = new double[10]; ❷
```
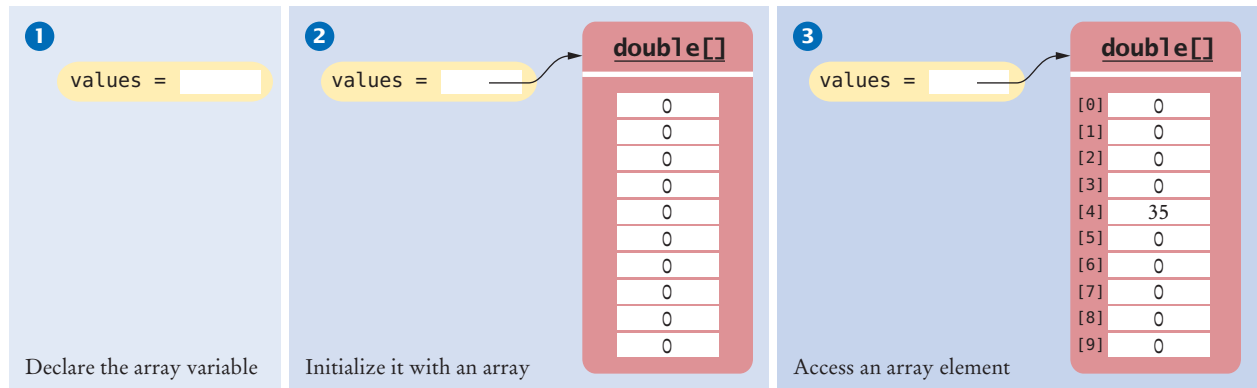
Figure 1 An Array of Size 10

Now `values` is initialized with an array of 10 numbers. By default, each number in the array is 0.

When you declare an array, you can specify the initial values. For example,

```
double[] moreValues = { 32, 54, 67.5, 29, 35, 80, 115, 44.5, 100, 65 };
```

When you supply initial values, you don't use the `new` operator. The compiler determines the length of the array by counting the initial values.

To access a value in an array, you specify which "slot" you want to use. That is done with the `[]` operator:

```
values[4] = 35;  ❸
```

Now the number 4 slot of `values` is filled with 35 (see Figure 1). This "slot number" is called an *index*. Each slot in an array contains an *element*.

Because `values` is an array of `double` values, each element `values[i]` can be used like any variable of type `double`. For example, you can display the element with index 4 with the following command:

```
System.out.println(values[4]);
```

> 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.

Syntax 7.1   Arrays

Syntax   To construct an array: `new typeName[length]`

To access an element: *arrayReference*[*index*]

```
                                                       Element
                                                       type    Length
      Name of array variable
Type of array variable ⟍
                        double[] values = new double[10];

                        double[] moreValues = { 32, 54, 67.5, 29, 35 };

Use brackets to access an element.                            List of initial values

                        values[i] = 0;

                        The index must be ≥ 0 and < the length of the array.
                        See Common Error 7.1.
```

Before continuing, we must take care of an important detail of Java arrays. If you look carefully at Figure 1, you will find that the *fifth* element was filled when we changed `values[4]`. In Java, the elements of arrays are numbered *starting at 0*. That is, the legal elements for the `values` array are

© Luckie8/iStockphoto.

Like a post office box that is identified by a box number, an array element is identified by an index.

  `values[0]`, the first element
  `values[1]`, the second element
  `values[2]`, the third element
  `values[3]`, the fourth element
  `values[4]`, the fifth element
  . . .
  `values[9]`, the tenth element

In other words, the declaration

```
double[] values = new double[10];
```

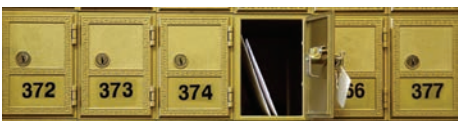creates an array with ten elements. In this array, an index can be any integer ranging from 0 to 9.

You have to be careful that the index stays within the valid range. Trying to access an element that does not exist in the array is a serious error. For example, if `values` has ten elements, you are not allowed to access `values[20]`. Attempting to access an element whose index is not within the valid index range is called a **bounds error**. The compiler does not catch this type of error. When a bounds error occurs at run time, it causes a run-time exception.

Here is a very common bounds error:

```
double[] values = new double[10];
values[10] = value;
```

There is no `values[10]` in an array with ten elements—the index can range from 0 to 9.

To avoid bounds errors, you will want to know how many elements are in an array. The expression `values.length` yields the length of the `values` array. Note that there are no parentheses following `length`.

> 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.

Table 1 Declaring Arrays

`int[] numbers = new int[10];`	An array of ten integers. All elements are initialized with zero.
`final int LENGTH = 10;` `int[] numbers = new int[LENGTH];`	It is a good idea to use a named constant instead of a "magic number".
`int length = in.nextInt();` `double[] data = new double[length];`	The length need not be a constant.
`int[] squares = { 0, 1, 4, 9, 16 };`	An array of five integers, with initial values.
`String[] friends = { "Emily", "Bob", "Cindy" };`	An array of three strings.
🚫 `double[] data = new int[10];`	**Error:** You cannot initialize a `double[]` variable with an array of type `int[]`.

Use the expression *array*.length to find the number of elements in an array.

The following code ensures that you only access the array when the index variable i is within the legal bounds:

```
if (0 <= i && i < values.length) { values[i] = value; }
```

Arrays suffer from a significant limitation: *their length is fixed*. If you start out with an array of 10 elements and later decide that you need to add additional elements, then you need to make a new array and copy all elements of the existing array into the new array. We will discuss this process in detail in Section 7.3.9.

To visit all elements of an array, use a variable for the index. Suppose values has ten elements and the integer variable i is set to 0, 1, 2, and so on, up to 9. Then the expression values[i] yields each element in turn. For example, this loop displays all elements in the values array:

```
for (int i = 0; i < 10; i++)
{
    System.out.println(values[i]);
}
```

Note that in the loop condition the index is *less than* 10 because there is no element corresponding to values[10].

7.1.2 Array References

An array reference specifies the location of an array. Copying the reference yields a second reference to the same array.

If you look closely at Figure 1, you will note that the variable values does not store any numbers. Instead, the array is stored elsewhere and the values variable holds a **reference** to the array. (The reference denotes the location of the array in memory.) You have already seen this behavior with objects in Section 2.8. When you access an object or array, you need not be concerned about the fact that Java uses references. This only becomes important when you copy a reference.

When you copy an array variable into another, both variables refer to the same array (see Figure 2).

```
int[] scores = { 10, 9, 7, 4, 5 };
int[] values = scores; // Copying array reference
```

You can modify the array through either of the variables:

```
scores[3] = 10;
System.out.println(values[3]); // Prints 10
```

Section 7.3.9 shows how you can make a copy of the *contents* of the array.
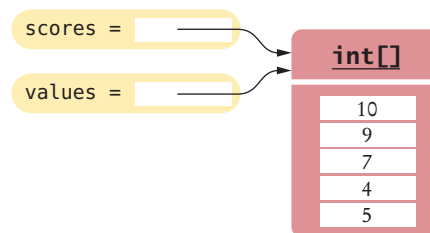

Figure 2
Two Array Variables Referencing the Same Array

7.1.3 Using Arrays with Methods

Arrays can occur as method arguments and return values.

Arrays can be method arguments and return values, just like any other values.

When you define a method with an array argument, you provide a parameter variable for the array. For example, the following method adds scores to a `Student` object:

```java
public void addScores(int[] values)
{
    for (int i = 0; i < values.length; i++)
    {
        totalScore = totalScore + values[i];
    }
}
```

To call this method, you have to provide an array:

```java
int[] scores = { 10, 9, 7, 10 };
fred.addScores(scores);
```

Conversely, a method can return an array. For example, a `Student` class can have a method

```java
public int[] getScores()
```

that returns an array with all of the student's scores.

7.1.4 Partially Filled Arrays

© Jarek Szymanski/iStockphoto.

With a partially filled array, you need to remember how many elements are filled.

An array cannot change size at run time. This is a problem when you don't know in advance how many elements you need. In that situation, you must come up with a good guess on the maximum number of elements that you need to store. For example, we may decide that we sometimes want to store more than ten elements, but never more than 100:

```java
final int LENGTH = 100;
double[] values = new double[LENGTH];
```

In a typical program run, only a part of the array will be occupied by actual elements. We call such an array a **partially filled array**. You must keep a *companion variable* that counts how many elements are actually used. In Figure 3 we call the companion variable `currentSize`.

The following loop collects inputs and fills up the `values` array:

```java
int currentSize = 0;
Scanner in = new Scanner(System.in);
while (in.hasNextDouble())
{
    if (currentSize < values.length)
    {
        values[currentSize] = in.nextDouble();
        currentSize++;
    }
}
```

With a partially filled array, keep a companion variable for the current size.

At the end of this loop, `currentSize` contains the actual number of elements in the array. Note that you have to stop accepting inputs if the `currentSize` companion variable reaches the array length.
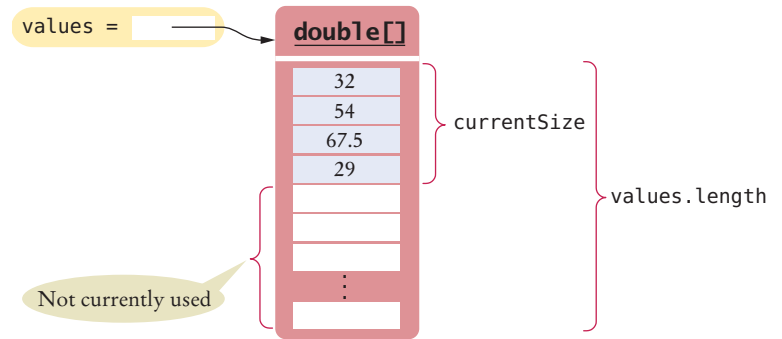
Figure 3
A Partially Filled
Array

To process the gathered array elements, you again use the companion variable, not the array length. This loop prints the partially filled array:

```
for (int i = 0; i < currentSize; i++)
{
   System.out.println(values[i]);
}
```

EXAMPLE CODE   See sec01 of your eText or companion code for a program that demonstrates array operations.

Common Error 7.1

Bounds Errors

Perhaps the most common error in using arrays is accessing a nonexistent element.

```
double[] values = new double[10];
values[10] = 5.4;
    // Error—values has 10 elements, and the index can range from 0 to 9
```

If your program accesses an array through an out-of-bounds index, there is no compiler error message. Instead, the program will generate an exception at run time.

Common Error 7.2

Uninitialized and Unfilled Arrays

A common error is to allocate an array variable, but not an actual array.

```
double[] values;
values[0] = 29.95; // Error—values not initialized
```

Array variables work exactly like object variables—they are only references to the actual array. To construct the actual array, you must use the new operator:

```
double[] values = new double[10];
```

Another common error is to allocate an array of objects and expect it to be filled with objects.

```
BankAccount[] accounts = new BankAccount[10]; // Contains ten null references
```

This array contains null references, not default bank accounts. You need to remember to fill the array, for example:

```
for (int i = 0; i < 10; i++)
{
   accounts[i] = new BankAccount();
}
```

Programming Tip 7.1

Use Arrays for Sequences of Related Items

Arrays are intended for storing sequences of values with the same meaning. For example, an array of test scores makes perfect sense:

```
int[] scores = new int[NUMBER_OF_SCORES];
```

But an array

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

that holds a person's age, bank balance, and shoe size in positions 0, 1, and 2 is bad design. It would be tedious for the programmer to remember which of these data values is stored in which array location. In this situation, it is far better to use three separate variables.

Programming Tip 7.2

Make Parallel Arrays into Arrays of Objects

Programmers who are familiar with arrays, but unfamiliar with object-oriented programming, sometimes distribute information across separate arrays. Here is a typical example: A program needs to manage bank data, consisting of account numbers and balances. Don't store the account numbers and balances in separate arrays.

```
// Don't do this
int[] accountNumbers;
double[] balances;
```

Arrays such as these are called **parallel arrays** (see Figure 4). The ith slice (accountNumbers[i] and balances[i]) contains data that need to be processed together.
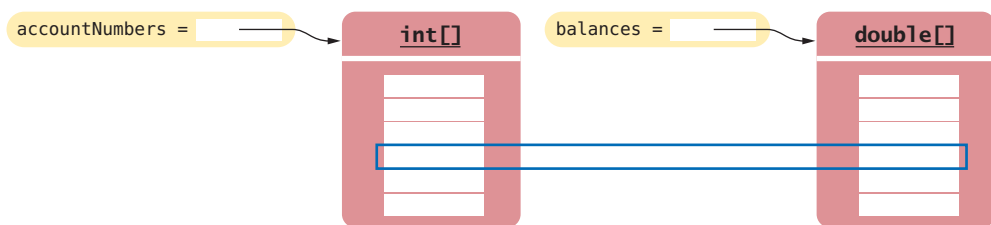
Figure 4   Avoid Parallel Arrays

Avoid parallel arrays by changing them into arrays of objects.

If you find yourself using two arrays that have the same length, ask yourself whether you couldn't replace them with a single array of a class type. Look at a slice and find the concept that it represents. Then make the concept into a class. In our example each slice contains an account number and a balance, describing a bank account. Therefore, it is an easy matter to use a single array of objects

```
BankAccount[] accounts;
```

(See Figure 5.)

Why is this beneficial? Think ahead. Maybe your program will change and you will need to store the owner of the bank account as well. It is a simple matter to update the BankAccount class. It may well be quite complicated to add a new array and make sure that all methods that accessed the original two arrays now also correctly access the third one.

Figure 5 Reorganizing Parallel Arrays into an Array of Objects

Special Topic 7.1

Methods with a Variable Number of Arguments

It is possible to declare methods that receive a variable number of arguments. For example, we can write a method that can add an arbitrary number of scores to a student:

```
fred.addScores(10, 7); // This method call has two arguments
fred.addScores(1, 7, 2, 9); // Another call to the same method, now with four arguments
```
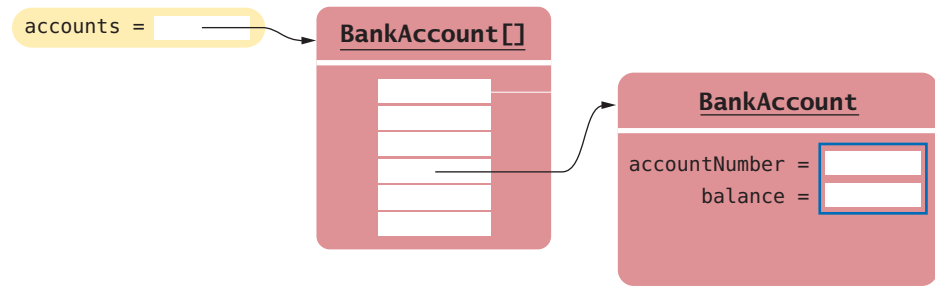
The method must be declared as

```
public void addScores(int... values)
```

The `int...` type indicates that the method can receive any number of `int` arguments. The `values` parameter variable is actually an `int[]` array that contains all arguments that were passed to the method.

The method implementation traverses the `values` array and processes the elements:

```
public void addScores(int... values)
{
   for (int i = 0; i < values.length; i++) // values is an int[]
   {
      totalScore = totalScore + values[i];
   }
}
```

Computing & Society 7.1   Computer Viruses

In November 1988, Robert Morris, a student at Cornell University, launched a so-called virus program that infected a significant fraction of computers connected to the Internet (which was much smaller then than it is now).

In order to attack a computer, a virus has to find a way to get its instructions executed. This particular program carried out a "buffer overrun" attack, providing an unexpectedly large input to a program on another machine. That program allocated an array of 512 characters, under the assumption that nobody would ever provide such a long input. Unfortunately, that program was written in the C programming language. C, unlike Java, does not check that an array index is less than the length of the array. If you write into an array using an index that is too large, you simply overwrite memory locations that belong to some other objects. C programmers are supposed to provide safety checks, but that had not happened in the program under attack. The virus program purposefully filled the 512-character array with 536 bytes. The excess 24 bytes overwrote a return address, which the attacker knew was stored just after the array. When the method that read the input was finished, it didn't return to its caller but to code supplied by the virus (see the figure). The virus was thus able to execute its code on a remote machine and infect it.

In Java, as in C, all programmers must be very careful not to overrun array boundaries. However, in Java, this error causes a run-time exception, and it never corrupts memory outside the array. This is one of the safety features of Java.

One may well speculate what would possess the virus author to spend weeks designing a program that disabled thousands of computers. It appears that the break-in was fully intended by the author, but the disabling of the computers was a bug caused by continuous reinfection. Morris was sentenced to three years probation, 400 hours of community service, and a $10,000 fine.

In recent years, computer attacks have intensified and the motives have become more sinister. Instead of disabling computers, viruses often take permanent residence in the attacked computers. Criminal enterprises rent out the processing power of millions of hijacked computers for sending spam e-mail or mining cryptocurrencies. Other viruses monitor every keystroke and send those that look like credit card numbers or banking passwords to their master.

Typically, a machine gets infected because a user executes code downloaded from the Internet, clicking on an icon or link that purports to be a game or video clip. Antivirus programs check all downloaded programs against an ever-growing list of known viruses.

When you use a computer for managing your finances, you need to be aware of the risk of infection. If a virus reads your banking password and empties your account, you will have a hard time convincing your financial institution that it wasn't your act, and you will most likely lose your money. Keep your operating system and antivirus program up to date, and don't click on suspicious links on a web page or in your e-mail inbox. Use banks that require "two-factor authentication" for major transactions, such as a callback on your cell phone.

Viruses are even used for military purposes. In 2010, a virus dubbed Stuxnet spread through Microsoft Windows and infected USB sticks. The virus looked for Siemens industrial computers and reprogrammed them in subtle ways. It appears that the virus was designed to damage the centrifuges of the Iranian nuclear enrichment operation. The computers controlling the centrifuges were not connected to the Internet, but they were configured with USB sticks, some of which were infected. Security researchers believe that the virus was developed by U.S. and Israeli intelligence agencies, and that it was successful in slowing down the Iranian nuclear program. Neither country has officially acknowledged or denied their role in the attacks.
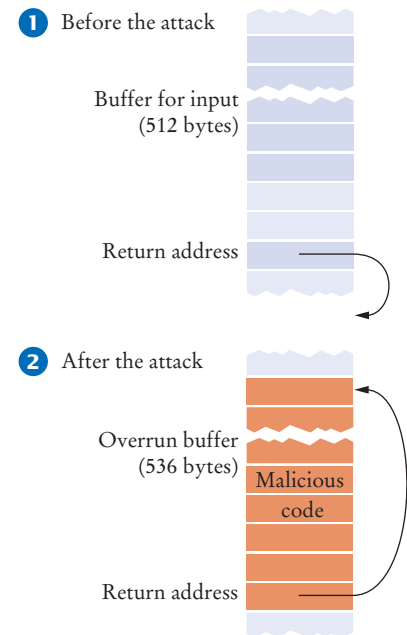
A "Buffer Overrun" Attack

7.2 The Enhanced for Loop

You can use the enhanced for loop to visit all elements of an array.

Often, you need to visit all elements of an array. The *enhanced* for *loop* makes this process particularly easy to program.

Here is how you use the enhanced for loop to total up all elements in an array named values:

```
double[] values = . . .;
double total = 0;
for (double element : values)
{
    total = total + element;
}
```

The loop body is executed for each element in the array values. At the beginning of each loop iteration, the next element is assigned to the variable element. Then the loop body is executed. You should read this loop as "for each element in values".

This loop is equivalent to the following for loop with an explicit index variable:

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

Note an important difference between the enhanced for loop and the basic for loop. In the enhanced for loop, the *element variable* is assigned values[0], values[1], and so on. In the basic for loop, the *index variable* i is assigned 0, 1, and so on.

Keep in mind that the enhanced for loop has a very specific purpose: getting the elements of a collection, from the beginning to the end. It is not suitable for all array algorithms. In particular, the enhanced for loop does not allow you to modify the contents of an array. The following loop does not fill an array with zeroes:

> Use the enhanced for loop if you do not need the index values in the loop body.

© Steve Cole/iStockphoto.

The enhanced for *loop is a convenient mechanism for traversing all elements in a collection.*

```
for (double element : values)
{
    element = 0; // ERROR: this assignment does not modify array elements
}
```

When the loop is executed, the variable element is set to values[0]. Then element is set to 0, then to values[1], then to 0, and so on. The values array is not modified. The remedy is simple: Use a basic for loop.

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

Syntax 7.2   The Enhanced for Loop

Syntax
```
for (typeName variable : collection)
{
    statements
}
```

This variable is set in each loop iteration. It is only defined inside the loop.

An array

```
for (double element : values)
{
    sum = sum + element;
}
```

These statements are executed for each element.

The variable contains an element, not an index.

EXAMPLE CODE   See sec02 of your eText or companion code for a program that demonstrates the enhanced for loop.

7.3 Common Array Algorithms

In the following sections, we discuss some of the most common algorithms for working with arrays. If you use a partially filled array, remember to replace `values.length` with the companion variable that represents the current size of the array.

7.3.1 Filling

This loop fills an array with squares (0, 1, 4, 9, 16, ...). Note that the element with index 0 contains 0^2, the element with index 1 contains 1^2, and so on.

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

7.3.2 Sum and Average Value

You have already encountered this algorithm in Section 6.7.1. When the values are located in an array, the code looks much simpler:

```
double total = 0;
for (double element : values)
{
   total = total + element;
}
double average = 0;
if (values.length > 0) { average = total / values.length; }
```

7.3.3 Maximum and Minimum

Use the algorithm from Section 6.7.5 that keeps a variable for the largest element already encountered. Here is the implementation of that algorithm for an array:

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

© CEFutcher/iStockphoto.

Note that the loop starts at 1 because we initialize `largest` with `values[0]`.
   To compute the smallest element, reverse the comparison.
   These algorithms require that the array contain at least one element.

7.3.4 Element Separators

When you display the elements of an array, you usually want to separate them, often with commas or vertical lines, like this:

```
32 | 54 | 67.5 | 29 | 35
```

When separating elements, don't place a separator before the first element.

Note that there is one fewer separator than there are numbers. Print the separator before each element in the sequence *except the initial one* (with index 0) like this:

```java
for (int i = 0; i < values.length; i++)
{
   if (i > 0)
   {
      System.out.print(" | ");
   }
   System.out.print(values[i]);
}
```

© trutenka/iStockphoto.

To print five elements, you need four separators.

If you want comma separators, you can use the `Arrays.toString` method. (You'll need to import `java.util.Arrays`.) The expression

```java
Arrays.toString(values)
```

returns a string describing the contents of the array `values` in the form

```
[32, 54, 67.5, 29, 35]
```

The elements are surrounded by a pair of brackets and separated by commas. This method can be convenient for debugging:

```java
System.out.println("values=" + Arrays.toString(values));
```

7.3.5 Linear Search

You often need to search for the position of a specific element in an array so that you can replace or remove it. Visit all elements until you have found a match or you have come to the end of the array. Here we search for the position of the first element in an array that is equal to 100:

© yekorzh/Getty Images.

To search for a specific element, visit the elements and stop when you encounter the match.

```java
int searchedValue = 100;
int pos = 0;
boolean found = false;
while (pos < values.length && !found)
{
   if (values[pos] == searchedValue)
   {
      found = true;
   }
   else
   {
      pos++;
   }
}
if (found) { System.out.println("Found at position: " + pos); }
else { System.out.println("Not found"); }
```

A linear search inspects elements in sequence until a match is found.

This algorithm is called **linear search** or *sequential search* because you inspect the elements in sequence. If the array is sorted, you can use the more efficient **binary search** algorithm. We discuss binary search in Chapter 14.

7.3.6 Removing an Element

Suppose you want to remove the element with index pos from the array values. As explained in Section 7.1.4, you need a companion variable for tracking the number of elements in the array. In this example, we use a companion variable called currentSize.

If the elements in the array are not in any particular order, simply overwrite the element to be removed with the *last* element of the array, then decrement the current-Size variable. (See Figure 6.)

```
values[pos] = values[currentSize - 1];
currentSize--;
```
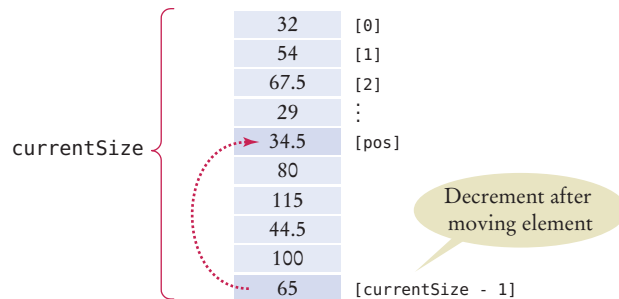

Figure 6 Removing an Element in an Unordered Array

The situation is more complex if the order of the elements matters. Then you must move all elements following the element to be removed to a lower index, and then decrement the variable holding the size of the array. (See Figure 7.)

```
for (int i = pos + 1; i < currentSize; i++)
{
   values[i - 1] = values[i];
}
currentSize--;
```
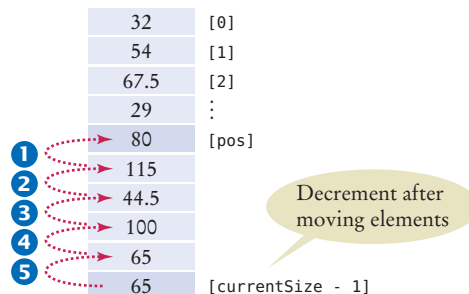

Figure 7 Removing an Element in an Ordered Array

7.3.7 Inserting an Element

In this section, you will see how to insert an element into an array. Note that you need a companion variable for tracking the array size, as explained in Section 7.1.4.

If the order of the elements does not matter, you can simply insert new elements at the end, incrementing the variable tracking the size.

```
if (currentSize < values.length)
{
   currentSize++;
   values[currentSize - 1] = newElement;
}
```
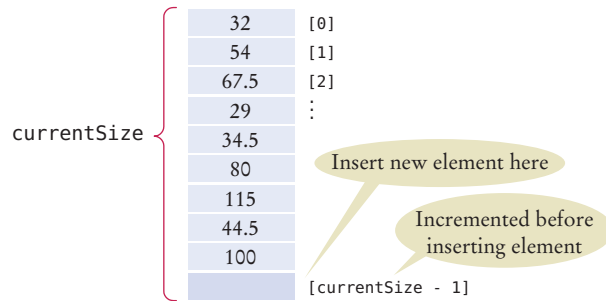

Figure 8 Inserting an Element in an Unordered Array

It is more work to insert an element at a particular position in the middle of an array. First, move all elements after the insertion location to a higher index. Then insert the new element (see Figure 9).
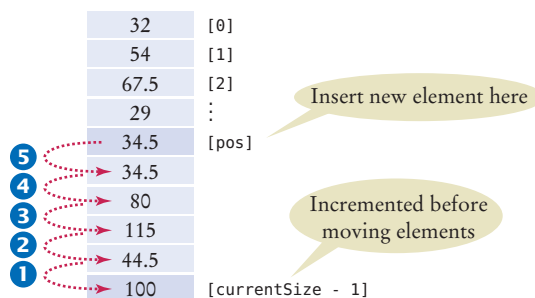
Figure 9 Inserting an Element in an Ordered Array

Note the order of the movement: When you remove an element, you first move the next element to a lower index, then the one after that, until you finally get to the end of the array. When you insert an element, you start at the end of the array, move that element to a higher index, then move the one before that, and so on until you finally get to the insertion location.

Before inserting an element, move elements to the end of the array *starting with the last one.*

```
if (currentSize < values.length)
{
   currentSize++;
   for (int i = currentSize - 1; i > pos; i--)
   {
      values[i] = values[i - 1];
   }
   values[pos] = newElement;
}
```

7.3.8 Swapping Elements

You often need to swap elements of an array. For example, you can sort an array by repeatedly swapping elements that are not in order.

Consider the task of swapping the elements at positions i and j of an array values. We'd like to set values[i] to values[j]. But that overwrites the value that is currently stored in values[i], so we want to save that first:

> Use a temporary variable when swapping two elements.

```
double temp = values[i];  ❷
values[i] = values[j];  ❸
```

Now we can set values[j] to the saved value.

```
values[j] = temp;  ❹
```

Figure 10 shows the process.
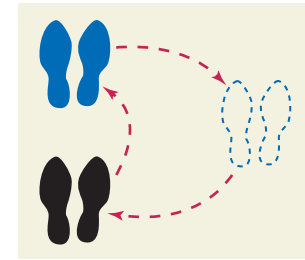
To swap two elements, you need a temporary variable.
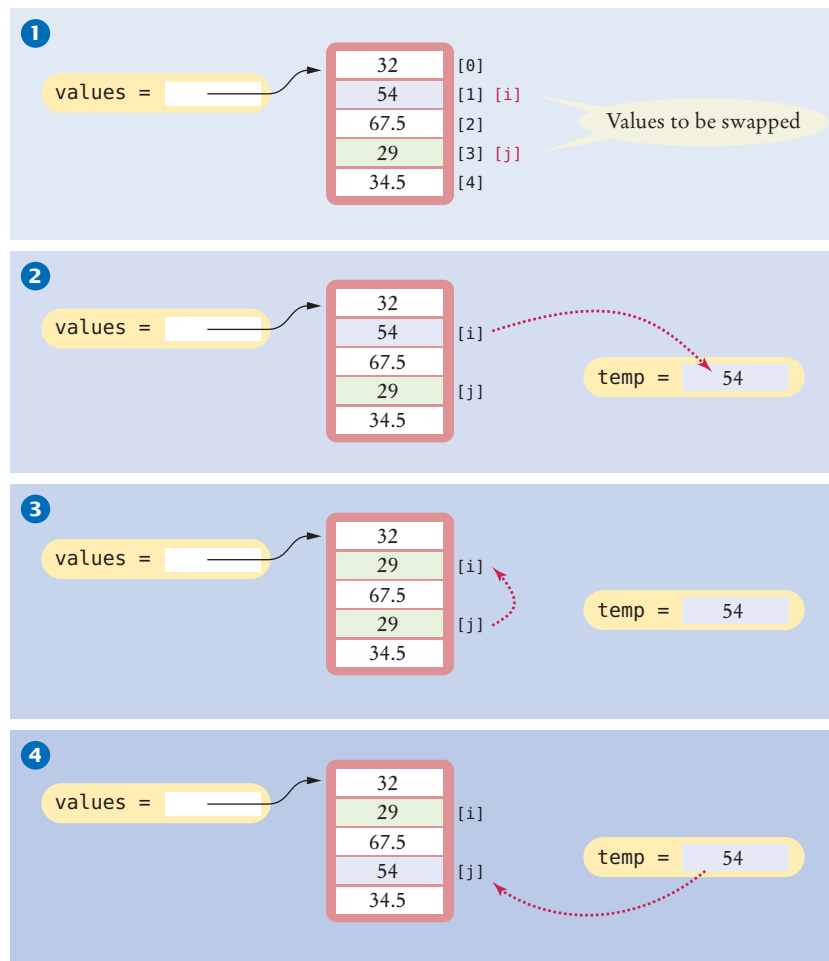
Figure 10 Swapping Array Elements

7.3.9 Copying Arrays

Array variables do not themselves hold array elements. They hold a reference to the actual array. If you copy the reference, you get another reference to the same array (see Figure 11):

```
double[] values = new double[6];
. . .  // Fill array
double[] prices = values;  ❶
```

Use the Arrays. copyOf method to copy the elements of an array into a new array.

If you want to make a true copy of an array, call the Arrays.copyOf method (as shown in Figure 11).

```
double[] prices = Arrays.copyOf(values, values.length);  ❷
```

The call Arrays.copyOf(values, n) allocates an array of length n, copies the first n elements of values (or the entire values array if n > values.length) into it, and returns the new array.
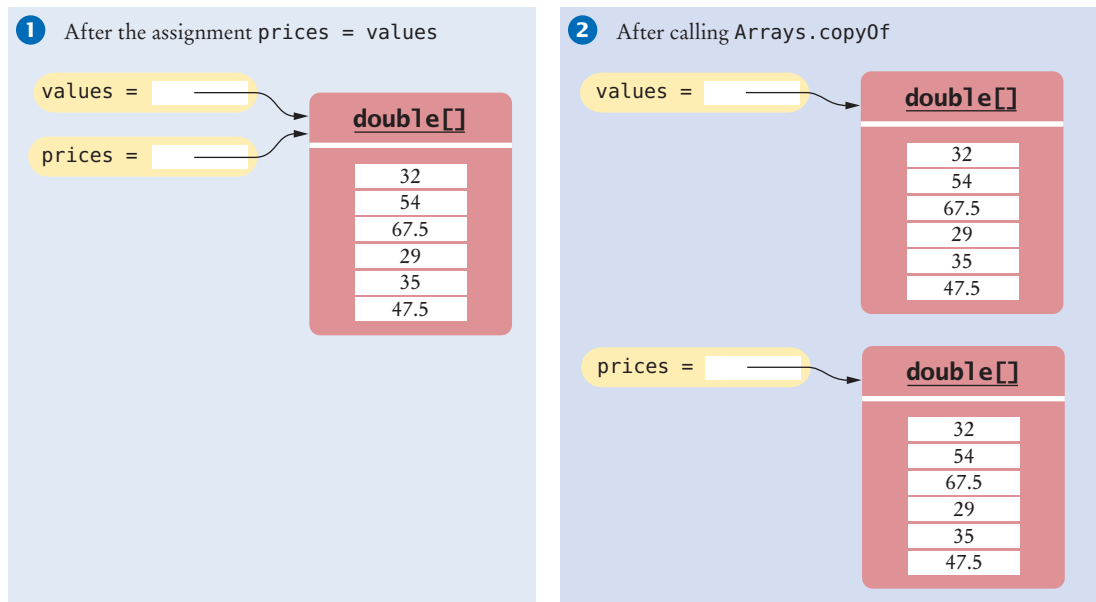
Figure 11   Copying an Array Reference versus Copying an Array

In order to use the Arrays class, you need to add the following statement to the top of your program:

```
import java.util.Arrays;
```

Another use for Arrays.copyOf is to grow an array that has run out of space. The following statements have the effect of doubling the length of an array (see Figure 12):

```
double[] newValues = Arrays.copyOf(values, 2 * values.length);  ❶
values = newValues;  ❷
```

1 Move elements to a larger array

```
values =
```
double[]
```
0
0
0
0
0
0
```

```
newValues =
```
double[]
```
0
0
0
0
0
0
```

2 Store the reference to the larger array in `values`

```
values =
```
double[]
```
0
0
0
0
0
0
```

```
newValues =
```
double[]
```
0
0
0
0
0
0
```
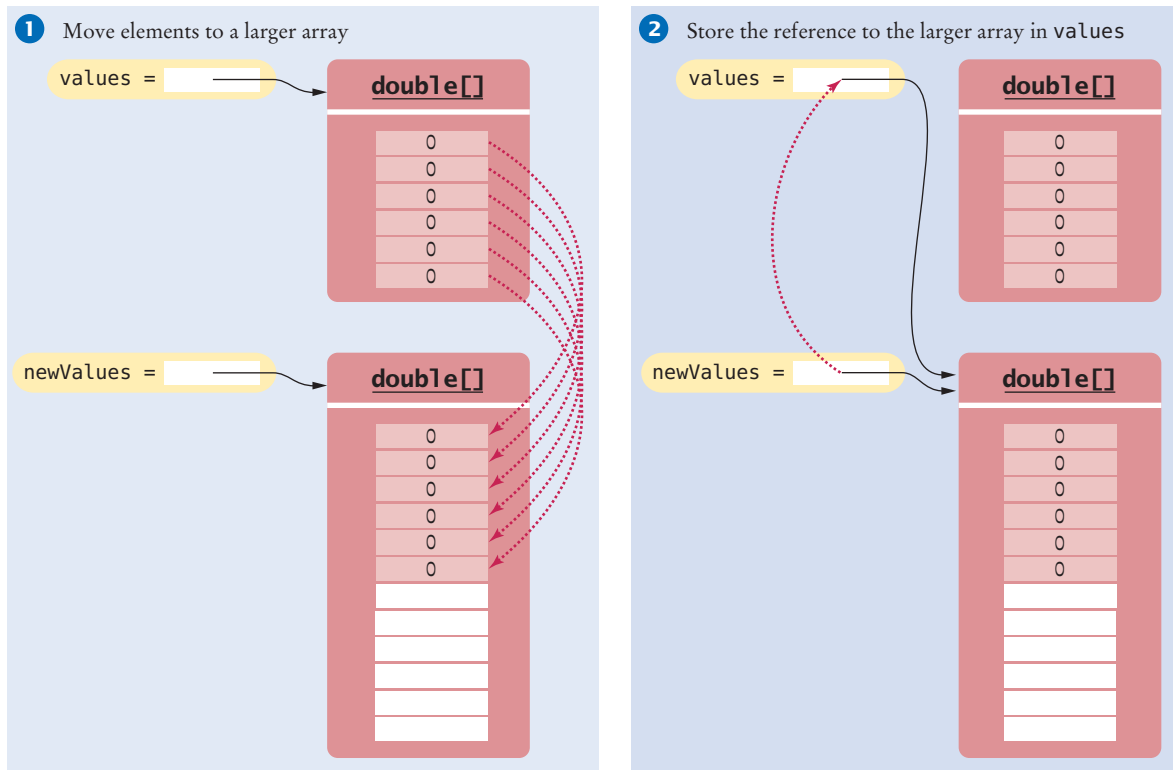
Figure 12   Growing an Array

7.3.10 Reading Input

If you know how many inputs the user will supply, it is simple to place them into an array:

```
double[] inputs = new double[NUMBER_OF_INPUTS];
for (i = 0; i < inputs.length; i++)
{
   inputs[i] = in.nextDouble();
}
```

However, this technique does not work if you need to read a sequence of arbitrary length. In that case, add the inputs to an array until the end of the input has been reached.

```
int currentSize = 0;
while (in.hasNextDouble() && currentSize < inputs.length)
{
   inputs[currentSize] = in.nextDouble();
   currentSize++;
}
```

Now `inputs` is a partially filled array, and the companion variable `currentSize` is set to the number of inputs.

However, this loop silently throws away inputs that don't fit into the array. A better approach is to grow the array to hold all inputs.

```
double[] inputs = new double[INITIAL_SIZE];
int currentSize = 0;
while (in.hasNextDouble())
{
   // Grow the array if it has been completely filled
   if (currentSize >= inputs.length)
   {
      inputs = Arrays.copyOf(inputs, 2 * inputs.length); // Grow the inputs array
   }

   inputs[currentSize] = in.nextDouble();
   currentSize++;
}
```

When you are done, you can discard any excess (unfilled) elements:

```
inputs = Arrays.copyOf(inputs, currentSize);
```

The following program puts these algorithms to work, solving the task that we set ourselves at the beginning of this chapter: to mark the largest value in an input sequence.

sec03/LargestInArray.java

```
1  import java.util.Scanner;
2
3  /**
4     This program reads a sequence of values and prints them,
5     marking the largest value.
6  */
7  public class LargestInArray
8  {
9     public static void main(String[] args)
10    {
11       final int LENGTH = 100;
12       double[] values = new double[LENGTH];
13       int currentSize = 0;
14
15       // Read inputs
16
17       System.out.println("Please enter values, Q to quit:");
18       Scanner in = new Scanner(System.in);
19       while (in.hasNextDouble() && currentSize < values.length)
20       {
21          values[currentSize] = in.nextDouble();
22          currentSize++;
23       }
24
25       // Find the largest value
26
27       double largest = values[0];
28       for (int i = 1; i < currentSize; i++)
29       {
30          if (values[i] > largest)
31          {
32             largest = values[i];
33          }
34       }
35
36       // Print all values, marking the largest
37
```

```
38        for (int i = 0; i < currentSize; i++)
39        {
40           System.out.print(values[i]);
41           if (values[i] == largest)
42           {
43              System.out.print(" <== largest value");
44           }
45           System.out.println();
46        }
47     }
48 }
```

Program Run

```
Please enter values, Q to quit:
34.5 80 115 44.5 Q
34.5
80
115 <== largest value
44.5
```

Common Error 7.3

Underestimating the Size of a Data Set

Programmers commonly underestimate the amount of input data that a user will pour into an unsuspecting program. Suppose you write a program to search for text in a file. You store each line in a string, and keep an array of strings. How big do you make the array? Surely nobody is going to challenge your program with an input that is more than 100 lines. Really? It is very easy to feed in the entire text of *Alice in Wonderland* or *War and Peace* (which are available on the Internet). All of a sudden, your program has to deal with tens or hundreds of thousands of lines. You either need to allow for large inputs or politely reject the excess input.

Special Topic 7.2

Sorting with the Java Library

Sorting an array efficiently is not an easy task. You will learn in Chapter 14 how to implement efficient sorting algorithms. Fortunately, the Java library provides an efficient sort method.

To sort an array values, call

```
Arrays.sort(values);
```

If the array is partially filled, call

```
Arrays.sort(values, 0, currentSize);
```

© ProstoVova/iStockphoto.

7.4 Problem Solving: Adapting Algorithms

In Section 7.3, you were introduced to a number of fundamental array algorithms. These algorithms form the building blocks for many programs that process arrays. In general, it is a good problem-solving strategy to have a repertoire of fundamental algorithms that you can combine and adapt.

By combining fundamental algorithms, you can solve complex programming tasks.

Consider this example problem: You are given the quiz scores of a student. You are to compute the final quiz score, which is the sum of all scores after dropping the lowest one. For example, if the scores are

```
8   7   8.5   9.5   7   4   10
```

then the final score is 50.

We do not have a ready-made algorithm for this situation. Instead, consider which algorithms may be related. These include:

- Calculating the sum (Section 7.3.2)
- Finding the minimum value (Section 7.3.3)
- Removing an element (Section 7.3.6)

We can formulate a plan of attack that combines these algorithms:

Find the minimum.
Remove the minimum from the array.
Calculate the sum.

Let's try it out with our example. The minimum of

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
	8	7	8.5	9.5	7	4	10

is 4. How do we remove it?

Now we have a problem. The removal algorithm in Section 7.3.6 locates the element to be removed by using the *position* of the element, not the value.

But we have another algorithm for that:

- Linear search (Section 7.3.5)

We need to fix our plan of attack:

Find the minimum value.
Find the position of the minimum.
Remove the element at the position.
Calculate the sum.

Will it work? Let's continue with our example.

We found a minimum value of 4. Linear search tells us that the value 4 occurs at position 5.

	[0]	[1]	[2]	[3]	[4]	[5]	[6]
	8	7	8.5	9.5	7	4	10

We remove it:

	[0]	[1]	[2]	[3]	[4]	[5]
	8	7	8.5	9.5	7	10

Finally, we compute the sum: $8 + 7 + 8.5 + 9.5 + 7 + 10 = 50$.

This walkthrough demonstrates that our strategy works.

Can we do better? It seems a bit inefficient to find the minimum and then make another pass through the array to obtain its position.

We can adapt the algorithm for finding the minimum to yield the position of the minimum.

You should be familiar with the implementation of fundamental algorithms so that you can adapt them.

Here is the original algorithm:

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

When we find the smallest value, we also want to update the position:

```
if (values[i] < smallest)
{
   smallest = values[i];
   smallestPosition = i;
}
```

In fact, then there is no reason to keep track of the smallest value any longer. It is simply values[smallestPosition]. With this insight, we can adapt the algorithm as follows:

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

With this adaptation, our problem is solved with the following strategy:

Find the position of the minimum.
Remove the element at the position.
Calculate the sum.

The next section shows you a technique for discovering a new algorithm when none of the fundamental algorithms can be adapted to a task.

EXAMPLE CODE   See sec04 of your eText or companion code for a program that uses the adapted algorithm for finding the minimum.

HOW TO 7.1
Working with Arrays

In many data processing situations, you need to process a sequence of values. This How To walks you through the steps for storing input values in an array and carrying out computations with the array elements.

Problem Statement Consider again the problem from Section 7.4: A final quiz score is computed by adding all the scores, except for the lowest one. For example, if the scores are

    8 7 8.5 9.5 7 5 10

then the final score is 50.

Thierry Dosogne/The Image Bank/Getty Images, Inc.

Step 1 Decompose your task into steps.

You will usually want to break down your task into multiple steps, such as

- Reading the data into an array.
- Processing the data in one or more steps.
- Displaying the results.

When deciding how to process the data, you should be familiar with the array algorithms in Section 7.3. Most processing tasks can be solved by using one or more of these algorithms.

In our sample problem, we will want to read the data. Then we will remove the minimum and compute the total. For example, if the input is 8 7 8.5 9.5 7 5 10, we will remove the minimum of 5, yielding 8 7 8.5 9.5 7 10. The sum of those values is the final score of 50.

Thus, we have identified three steps:

Read inputs.
Remove the minimum.
Calculate the sum.

Step 2 Determine which algorithm(s) you need.

Sometimes, a step corresponds to exactly one of the basic array algorithms in Section 7.3. That is the case with calculating the sum (Section 7.3.2) and reading the inputs (Section 7.3.10). At other times, you need to combine several algorithms. To remove the minimum value, you can find the minimum value (Section 7.3.3), find its position (Section 7.3.5), and remove the element at that position (Section 7.3.6).

We have now refined our plan as follows:

Read inputs.
Find the minimum.
Find the position of the minimum.
Remove the element at the position.
Calculate the sum.

This plan will work—see Section 7.4. But here is an alternate approach. It is easy to compute the sum and subtract the minimum. Then we don't have to find its position. The revised plan is

Read inputs.
Find the minimum.
Calculate the sum.
Subtract the minimum from the sum.

Step 3 Use classes and methods to structure the program.

Even though it may be possible to put all steps into the main method, this is rarely a good idea. It is better to carry out each processing step in a separate method. It is also a good idea to come up with a class that is responsible for collecting and processing the data.

In our example, let's provide a class Student. A student has an array of scores.

```
public class Student
{
    private double[] scores;
    private double scoresSize;
    . . .
    public Student(int capacity) { . . . }
    public boolean addScore(double score) { . . . }
    public double finalScore() { . . . }
}
```

A second class, ScoreAnalyzer, is responsible for reading the user input and displaying the result. Its main method simply calls the Student methods:

```
Student fred = new Student(100);
System.out.println("Please enter values, Q to quit:");
while (in.hasNextDouble())
{
   if (!fred.addScore(in.nextDouble()))
   {
      System.out.println("Too many scores.");
      return;
   }
}
System.out.println("Final score: " + fred.finalScore());
```

Now the finalScore method must do the heavy lifting. It too should not have to do all the work. Instead, we will supply helper methods

```
public double sum()
public double minimum()
```

These methods simply implement the algorithms in Sections 7.3.2 and 7.3.3.
  Then the finalScore method becomes

```
public double finalScore()
{
   if (scoresSize == 0)
   {
      return 0;
   }
   else
   {
      return sum() - minimum();
   }
}
```

Step 4   Assemble and test the program.

Place your methods into a class. Review your code and check that you handle both normal and exceptional situations. What happens with an empty array? One that contains a single element? When no match is found? When there are multiple matches? Consider these boundary conditions and make sure that your program works correctly.
  In our example, it is impossible to compute the minimum if the array is empty. In that case, we should terminate the program with an error message *before* attempting to call the minimum method.
  What if the minimum value occurs more than once? That means that a student had more than one test with the same low score. We subtract only one of the occurrences of that low score, and that is the desired behavior.
  The following table shows test cases and their expected output:

Test Case	Expected Output	Comment
8 7 8.5 9.5 7 5 10	50	See Step 1.
8 7 7 9	24	Only one instance of the low score should be removed.
8	0	After removing the low score, no score remains.
(no inputs)	**Error**	That is not a legal input.

EXAMPLE CODE   See your eText or the how_to_1 folder of your companion code for the complete program.

WORKED EXAMPLE 7.1

Rolling the Dice

Learn how to analyze a set of die tosses to see whether the die is "fair". See your eText or visit `wiley.com/go/bjeo7`.

© ktsimage/iStockphoto.

7.5 Problem Solving: Discovering Algorithms by Manipulating Physical Objects

In Section 7.4, you saw how to solve a problem by combining and adapting known algorithms. But what do you do when none of the standard algorithms is sufficient for your task? In this section, you will learn a technique for discovering algorithms by manipulating physical objects.

Consider the following task: You are given an array whose size is an even number, and you are to switch the first and the second half. For example, if the array contains the eight numbers

| 9 | 13 | 21 | 4 | 11 | 7 | 1 | 3 |

then you should change it to

| 11 | 7 | 1 | 3 | 9 | 13 | 21 | 4 |

Many students find it quite challenging to come up with an algorithm. They may know that a loop is required, and they may realize that elements should be inserted (Section 7.3.7) or swapped (Section 7.3.8), but they do not have sufficient intuition to draw diagrams, describe an algorithm, or write down pseudocode.

One useful technique for discovering an algorithm is to manipulate physical objects. Start by lining up some objects to denote an array. Coins, playing cards, or small toys are good choices.

Here we arrange eight coins:

Use a sequence of coins, playing cards, or toys to visualize an array of values.

coins: © jamesbenet/iStockphoto; dollar coins: © JordiDelgado/iStockphoto.

Now let's step back and see what we can do to change the order of the coins.

Manipulating physical objects can give you ideas for discovering algorithms.

© JenCon/iStockphoto.

We can remove a coin (Section 7.3.6):
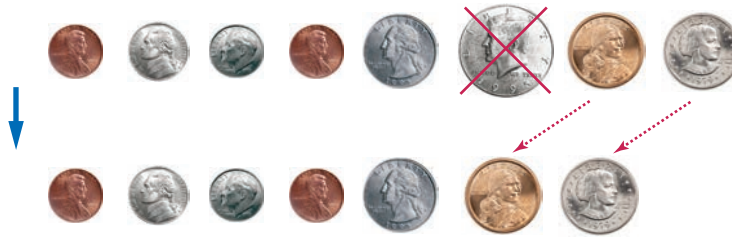
We can insert a coin (Section 7.3.7):

Or we can swap two coins (Section 7.3.8).

Go ahead—line up some coins and try out these three operations right now so that you get a feel for them.

Now how does that help us with our problem, switching the first and the second half of the array?

Let's put the first coin into place, by swapping it with the fifth coin. However, as Java programmers, we will say that we swap the coins in positions 0 and 4:

Next, we swap the coins in positions 1 and 5:

Two more swaps, and we are done:

Now an algorithm is becoming apparent:

```
i = 0
j = … // We'll think about that in a minute
While // Don't know yet
    Swap elements at positions i and j.
    i++
    j++
```

Where does the variable j start? When we have eight coins, the coin at position zero is moved to position 4. In general, it is moved to the middle of the array, or to position *size / 2*.

And how many iterations do we make? We need to swap all coins in the first half. That is, we need to swap *size / 2* coins.

The pseudocode is

```
i = 0
j = size / 2
While i < size / 2
    Swap elements at positions i and j.
    i++
    j++
```

It is a good idea to make a walkthrough of the pseudocode (see Section 6.2). You can use paper clips to denote the positions of the variables i and j. If the walkthrough is successful, then we know that there was no "off-by-one" error in the pseudocode. Exercise • E7.9 asks you to translate the pseudocode to Java. Exercise •• R7.25 suggests a different algorithm for switching the two halves of an array, by repeatedly removing and inserting coins.

You can use paper clips as position markers or counters.

Many people find that the manipulation of physical objects is less intimidating than drawing diagrams or mentally envisioning algorithms. Give it a try when you need to design a new algorithm!

EXAMPLE CODE See sec05 of your eText or companion code for a program that implements the algorithm that switches the first and second halves of an array.

7.6 Two-Dimensional Arrays

It often happens that you want to store collections of values that have a two-dimensional layout. Such data sets commonly occur in financial and scientific applications. An arrangement consisting of rows and columns of values is called a **two-dimensional array**, or a *matrix*.

Let's explore how to store the example data shown in Figure 13: the medal counts of the figure skating competitions at the 2014 Winter Olympics.

© Trub/iStockphoto.

	Gold	Silver	Bronze
Canada	0	3	0
Italy	0	0	1
Germany	0	0	1
Japan	1	0	0
Kazakhstan	0	0	1
Russia	3	1	1
South Korea	0	1	0
United States	1	0	1

© technotr/iStockphoto.

Figure 13 Figure Skating Medal Counts

7.6.1 Declaring Two-Dimensional Arrays

Use a two-dimensional array to store tabular data.

In Java, you obtain a two-dimensional array by supplying the number of rows and columns. For example, `new int[7][3]` is an array with seven rows and three columns. You store a reference to such an array in a variable of type `int[][]`. Here is a complete declaration of a two-dimensional array, suitable for holding our medal count data:

```
final int COUNTRIES = 8;
final int MEDALS = 3;
int[][] counts = new int[COUNTRIES][MEDALS];
```

Syntax 7.3 Two-Dimensional Array Declaration

Name *Element type* *Number of rows* *Number of columns*

```
double[][] tableEntries = new double[7][3];
```
All values are initialized with 0.

Name

List of initial values

```
int[][] data = {
                 { 16, 3, 2, 13 },
                 { 5, 10, 11, 8 },
                 { 9, 6, 7, 12 },
                 { 4, 15, 14, 1 },
             };
```

Alternatively, you can declare and initialize the array by grouping each row:

```
int[][] counts =
    {
        { 0, 3, 0 },
        { 0, 0, 1 },
        { 0, 0, 1 },
        { 1, 0, 0 },
        { 0, 0, 1 },
        { 3, 1, 1 },
        { 0, 1, 0 }
        { 1, 0, 1 }
    };
```

As with one-dimensional arrays, you cannot change the size of a two-dimensional array once it has been declared.

7.6.2 Accessing Elements

Individual elements in a two-dimensional array are accessed by using two index values, *array*[i][j].

To access a particular element in the two-dimensional array, you need to specify two index values in separate brackets to select the row and column, respectively (see Figure 14):

```
int medalCount = counts[3][1];
```
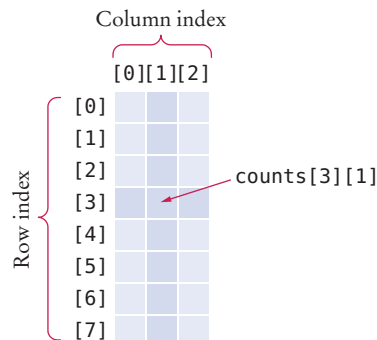

Figure 14
Accessing an Element in a
Two-Dimensional Array

To access all elements in a two-dimensional array, you use nested loops. For example, the following loop prints all elements of `counts`:

```
for (int i = 0; i < COUNTRIES; i++)
{
   // Process the ith row
   for (int j = 0; j < MEDALS; j++)
   {
      // Process the jth column in the ith row
      System.out.printf("%8d", counts[i][j]);
   }
   System.out.println(); // Start a new line at the end of the row
}
```

In these loops, the number of rows and columns were given as constants. Alternatively, you can use the following expressions:

- `counts.length` is the number of rows.
- `counts[0].length` is the number of columns. (See Special Topic 7.3 for an explanation of this expression.)

With these expressions, the nested loops become

```
for (int i = 0; i < counts.length; i++)
{
   for (int j = 0; j < counts[0].length; j++)
   {
      System.out.printf("%8d", counts[i][j]);
   }
   System.out.println();
}
```

7.6.3 Locating Neighboring Elements

Some programs that work with two-dimensional arrays need to locate the elements that are adjacent to an element. This task is particularly common in games. Figure 15 shows how to compute the index values of the neighbors of an element.

For example, the neighbors of `counts[3][1]` to the left and right are `counts[3][0]` and `counts[3][2]`. The neighbors to the top and bottom are `counts[2][1]` and `counts[4][1]`.

You need to be careful about computing neighbors at the boundary of the array. For example, `counts[0][1]` has no neighbor to the top. Consider the task of computing the sum of the neighbors to the top and bottom of the element `count[i][j]`. You need to check whether the element is located at the top or bottom of the array:

Figure 15 Neighboring Locations in a Two-Dimensional Array

```
int total = 0;
if (i > 0) { total = total + counts[i - 1][j]; }
if (i < ROWS - 1) { total = total + counts[i + 1][j]; }
```

7.6.4 Accessing Rows and Columns

You often need to access all elements in a row or column, for example to compute the sum of the elements or the largest element in a row or column. In our sample array, the row totals give us the total number of medals won by a particular country.

Finding the correct index values is a bit tricky, and it is a good idea to make a quick sketch. To compute the total of row i, we need to visit the following elements:

```
                    0            MEDALS - 1
```

```
row i  ——  [i][0] [i][1] [i][2]
```
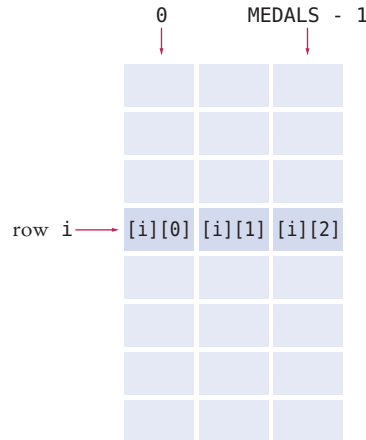
As you can see, we need to compute the sum of counts[i][j], where j ranges from 0 to MEDALS - 1. The following loop computes the total:

```
int total = 0;
for (int j = 0; j < MEDALS; j++)
{
    total = total + counts[i][j];
}
```

Computing column totals is similar. Form the sum of counts[i][j], where i ranges from 0 to COUNTRIES - 1.

```
int total = 0;
for (int i = 0; i < COUNTRIES; i++)
{
    total = total + counts[i][j];
}
```

```
                    column j
```

```
[0][j]    ——  0
[1][j]
[2][j]
[3][j]
[4][j]
[5][j]
[6][j]
[7][j]    ——  COUNTRIES - 1
```
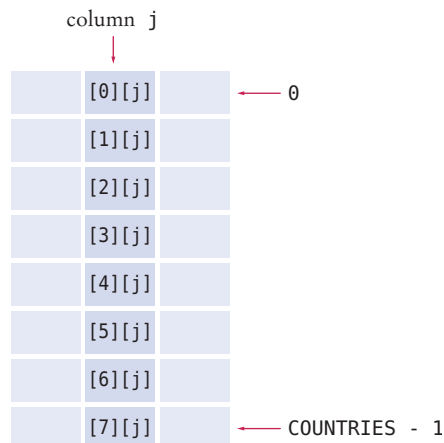
7.6.5 Two-Dimensional Array Parameters

When you use a two-dimensional array, you do not need to use separate variables for the dimensions of the array. If values is a two-dimensional array, then

- values.length is the number of rows.
- values[0].length is the number of columns. (See Special Topic 7.4 for an explanation of this expression.)

For example, the following method computes the sum of all elements in a two-dimensional array:

```java
public class Data
{
   private int[][] values;
   . . .

   public int sum()
   {
      int total = 0;
      for (int i = 0; i < values.length; i++)
      {
         for (int j = 0; j < values[0].length; j++)
         {
            total = total + values[i][j];
         }
      }
      return total;
   }
}
```

Working with two-dimensional arrays is illustrated in the following program. The program prints out the medal counts and the row totals.

sec06/Medals.java

```java
1  /**
2     This program prints a table of medal winner counts with row totals.
3  */
4  public class Medals
5  {
6     public static void main(String[] args)
7     {
8        final int COUNTRIES = 8;
9        final int MEDALS = 3;
10
11       String[] countries =
12          {
13             "Canada",
14             "Italy",
15             "Germany",
16             "Japan",
17             "Kazakhstan",
18             "Russia",
19             "South Korea",
20             "United States"
21          };
22
```

```
23        int[][] counts =
24           {
25              { 0, 3, 0 },
26              { 0, 0, 1 },
27              { 0, 0, 1 },
28              { 1, 0, 0 },
29              { 0, 0, 1 },
30              { 3, 1, 1 },
31              { 0, 1, 0 },
32              { 1, 0, 1 }
33           };
34
35        System.out.println("        Country    Gold  Silver  Bronze    Total");
36
37        // Print countries, counts, and row totals
38        for (int i = 0; i < COUNTRIES; i++)
39        {
40           // Process the ith row
41           System.out.printf("%15s", countries[i]);
42
43           int total = 0;
44
45           // Print each row element and update the row total
46           for (int j = 0; j < MEDALS; j++)
47           {
48              System.out.printf("%8d", counts[i][j]);
49              total = total + counts[i][j];
50           }
51
52           // Display the row total and print a new line
53           System.out.printf("%8d%n", total);
54        }
55     }
56  }
```

Program Run

Country	Gold	Silver	Bronze	Total
Canada	0	3	0	3
Italy	0	0	1	1
Germany	0	0	1	1
Japan	1	0	0	1
Kazakhstan	0	0	1	1
Russia	3	1	1	5
South Korea	0	1	0	1
United States	1	0	1	2

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. See your eText or visit wiley.com/go/bjeo7.

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 ith row, and the [j] operator selects the jth element in that row.

Figure 16 A Triangular Array
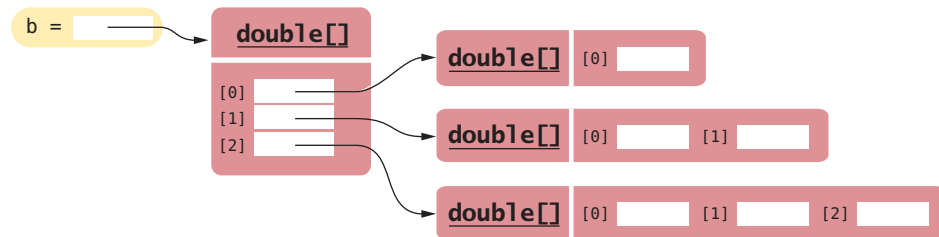
Note that the number of rows is b.length, and the length of the ith row is b[i].length. For example, the following pair of loops prints a ragged array:

```
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.

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. © Michael Brake/iStockphoto.

7.7.1 Declaring and Using Array Lists

The following statement declares an array list of strings:

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

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");

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

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

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

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"
```
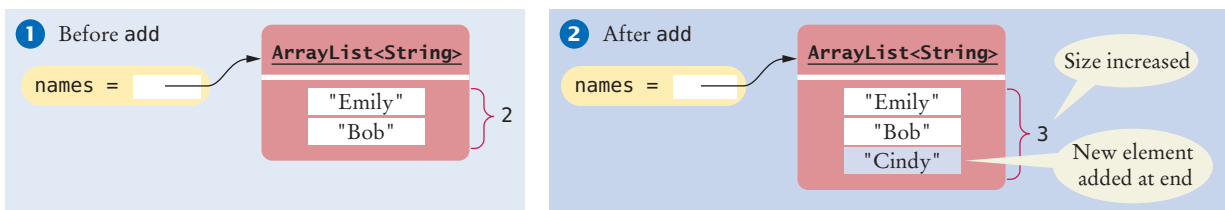
1 Before add

names =

ArrayList<String>
"Emily"
"Bob"
} 2

2 After add

names =

ArrayList<String>
"Emily"
"Bob"
"Cindy"
} 3

Size increased

New element added at end

Figure 17   Adding an Array List Element with add

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");
```

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).

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

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)`.
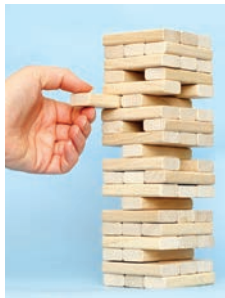
© Danijelm/iStockphoto.

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

1 Before `add`

names = → **ArrayList<String>**
"Emily"
"Bob"
"Carolyn"

2 After `names.add(1, "Ann")`

names = → **ArrayList<String>**
"Emily"
"Ann" ← New element added at index 1
"Bob" ← Moved from index 1 to 2
"Carolyn" ← Moved from index 2 to 3

3 After `names.remove(1)`

names = → **ArrayList<String>**
"Emily"
"Bob" ← Moved from index 2 to 1
"Carolyn" ← Moved from index 3 to 2

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

With an array list, it is very easy to get a quick printout. Simply pass the array list to the `println` method:
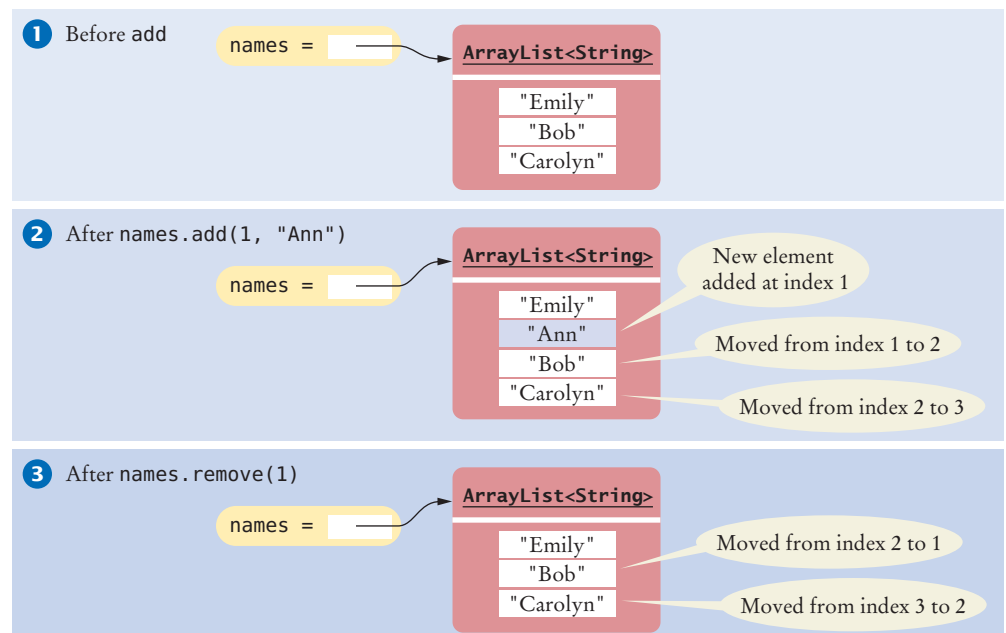
```
System.out.println(names); // Prints [Emily, Bob, Carolyn]
```

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);
}
```

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 of the array list.
`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.

7.7.3 Copying Array Lists

As with arrays, you need to remember that array list variables hold references. Copy-ing 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);
```
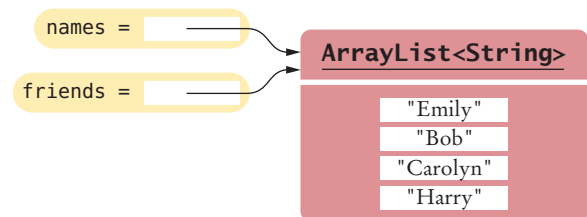
Figure 19
Copying an Array List Reference

7.7.4 Wrappers and Auto-boxing

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

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

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).

Figure 20 A Wrapper Class Variable

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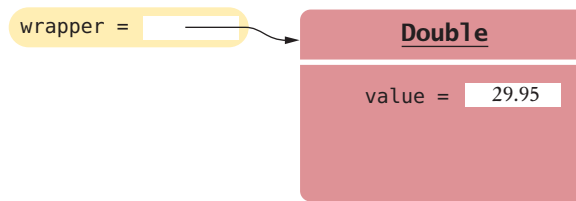
Because boxing and unboxing is automatic, you don't need to think about it. Simply remember to use the wrapper type when you declare array lists of numbers. From then on, use the primitive type and rely on auto-boxing.

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

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. © sandoclr/iStockphoto.

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 262). 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.

i	words
0	"Welcome", "to", "the", "island"
1	"Welcome", "the", "island"
2	

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.

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.

The following program shows how to mark the largest value in a sequence of values stored in an array list. Note how the program is an improvement over the array version in Section 7.3.10 on page 239. This program can process input sequences of arbitrary length.

sec07/LargestInArrayList.java

```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
```

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

There is 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.

For now, we will use the explicit syntax and include the type parameters with constructors. In later chapters, we will switch to the diamond syntax.

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**.

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

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

new version, you get that feedback. The process of checking each version of a pro-gram 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)
   {
      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 mul-tiple files, as in the following.

sec08/ScoreTester2.java

```
1  import java.util.Scanner;
2
3  public class ScoreTester2
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:

sec08/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.

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   Liability for Software Malfunction

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. 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 the 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.

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 for what constitutes a safe software design process.

Nowadays, we interact with vast numbers of computer-controlled machines that can endanger us. Self-driving vehicles are a good example. Overall, those machines may well make us safer, but the question remains: What risks are acceptable?
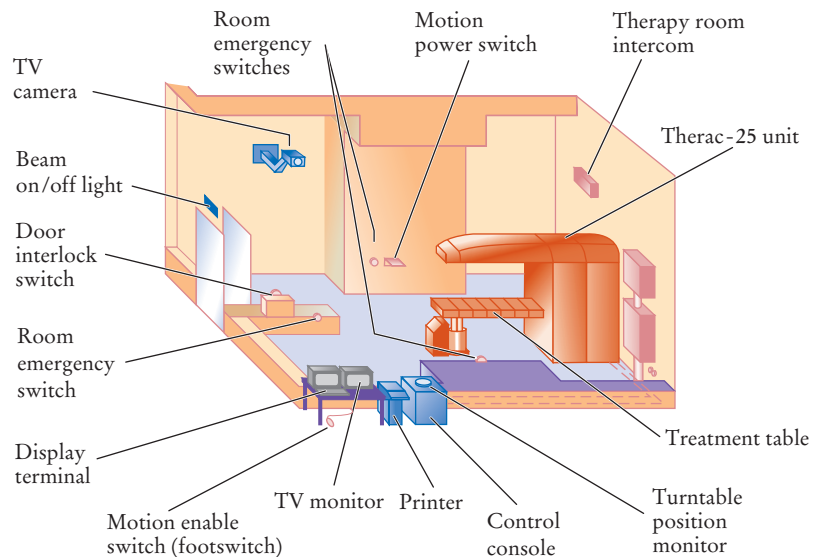
Typical Therac-25 Facility

CHAPTER SUMMARY

Use arrays for collecting values.
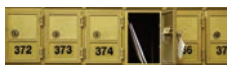
- An array collects a sequence of values of the same type.
- Individual elements in an array are accessed by an integer index i, using the notation *array*[i].
- An array element can be used like any variable.

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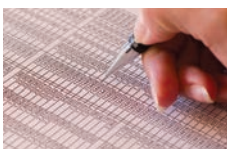
- Use a two-dimensional array to store tabular data.
- Individual elements in a two-dimensional array are accessed by using two index values, *array*[i][j].

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.util.ArrayList<E>
java.lang.Double	add
java.lang.Integer	get
java.util.Arrays	remove
copyOf	set
toString	size

•• **R7.1** Carry out the following tasks with an array:

    a. Allocate an array a of ten integers.

    b. Put the number 17 as the initial element of the array.

    c. Put the number 29 as the last element of the array.

    d. Fill the remaining elements with –1.

    e. Add 1 to each element of the array.

    f. Print all elements of the array, one per line.

    g. Print all elements of the array in a single line, separated by commas.

• **R7.2** Write a program that contains a bounds error. Run the program. What happens on your computer?

•• **R7.3** 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

•• **R7.4** 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; }
```

•• **R7.5** 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]; }`

■■■ **R7.6** 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.

■■ **R7.7** Write Java code for a loop that simultaneously computes both the maximum and minimum of an array.

■ **R7.8** 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;
}
```

■■ **R7.9** 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.

■■ **R7.10** 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++; }`

■■ **R7.11** 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; }
}
```

■ **R7.12** 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);
}
```

■■ **R7.13** 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.

■ **R7.14** 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.

■ **R7.15** 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.

■ **R7.16** 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.

■■ **R7.17** 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.

■■ **R7.18** 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:

2	3	5	7	11	13

3	5	7	11	13	2
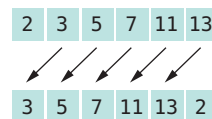
■■ **R7.19** Give pseudocode for an algorithm that removes all negative values from an array, preserving the order of the remaining elements.

■■ **R7.20** Suppose values is a *sorted* array of integers. Give pseudocode that describes how a new value can be inserted so that the resulting array stays sorted.

▪▪▪ **R7.21** 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.

▪▪▪ **R7.22** What is wrong with the following method that aims to fill an array with random numbers?

```java
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;
}
```

▪▪ **R7.23** You are given two arrays denoting *x*- and *y*-coordinates of a set of points in a 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?
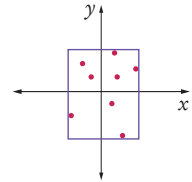
▪ **R7.24** Solve the quiz score problem described in Section 7.4 by sorting the array first. How do you need to modify the algorithm for computing the total?

▪▪ **R7.25** 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.

▪▪ **R7.26** 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.

▪▪ **R7.27** Write Java statements for performing the following tasks with an array declared as

```java
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.

▪▪ **R7.28** 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.

- **R7.29** 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.

- **R7.30** 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.

- **R7.31** 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.

- **R7.32** 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 R7.33** Define the terms regression testing and test suite.

- **Testing R7.34** What is the debugging phenomenon known as *cycling*? What can you do to avoid it?

PRACTICE EXERCISES

- **E7.1** Write a loop that reads ten numbers and a second loop that displays them in the opposite order from which they were entered.

- **E7.2** 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.

- **E7.3** Modify the `LargestInArray.java` program in Section 7.3.10 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

$$1 \quad 4 \quad 9 \quad 16 \quad 9 \quad 7 \quad 4 \quad 9 \quad 11$$

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

$$1 \quad 4 \quad 9 \quad 16 \quad 9 \quad 7 \quad 4 \quad 9 \quad 11$$

then the array is changed to

$$11 \quad 9 \quad 4 \quad 7 \quad 9 \quad 16 \quad 9 \quad 4 \quad 1$$

■■■ **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, 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 position in the second array.
  Remove the element at the position from the second array.
  Append the removed element to the permutation array.

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

■■ **E7.10** Write a class `DataSet` that stores a number of values of type `double`. Provide a constructor

```
public DataSet(int maximumNumberOfValues)
```

and a method

```
public void add(double value)
```

that adds a value, provided there is still room.

Provide methods to compute the sum, average, maximum, and minimum value.

■■ **E7.11** 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 to the right by one 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).

■■ **E7.12** 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; }
   public int get(int i) { return values[i]; }
   public int size() { return values.length; }
}
```

Add a method

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

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

■■ **E7.13** Add a method

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

to the Sequence class of Exercise •• E7.12 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.

■■■ **E7.14** Add a method

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

to the Sequence class of Exercise •• E7.12 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

$$1 \quad 4 \quad 9 \quad 16 \quad 9 \quad 7 \quad 4 \quad 9 \quad 11$$

is not a permutation of

$$11 \quad 11 \quad 7 \quad 9 \quad 16 \quad 4 \quad 1 \quad 4 \quad 9$$

You will probably need one or more helper methods.

E7.15 Add a method

```
public Sequence sum(Sequence other)
```

to the Sequence class of Exercise •• E7.12 that yields the sum of this sequence and another. If the sequences don't have the same length, assume that the missing elements are zero. For example, the sum of

$$1 \quad 4 \quad 9 \quad 16 \quad 9 \quad 7 \quad 4 \quad 9 \quad 11$$

and

$$11 \quad 11 \quad 7 \quad 9 \quad 16 \quad 4 \quad 1$$

is the sequence

$$12 \quad 15 \quad 16 \quad 25 \quad 25 \quad 11 \quad 5 \quad 9 \quad 11$$

E7.16 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.

E7.17 Add a method to the Table class below 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, include only the neighbors that are in the table. For example, if row and column are both 0, there are only three neighbors.

```
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; }
}
```

E7.18 Given the Table class of Exercise •• E7.17 , add a method that returns the sum of the ith row (if horizontal is true) or column (if horizontal is false):

```
public double sum(int i, boolean horizontal)
```

E7.19 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.

▪▪▪ E7.20 Repeat Exercise •• E7.19, but make the bars vertical, with the tallest bar twenty asterisks high.

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

▪▪▪ E7.21 Improve the program of Exercise •• E7.19 to work correctly when the data set contains negative values.

▪▪ E7.22 Improve the program of Exercise •• E7.19 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 **************
```

▪ E7.23 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

$$1 \quad 4 \quad 9 \quad 16$$

and b is the sequence

$$9 \quad 7 \quad 4 \quad 9 \quad 11$$

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

$$1 \quad 4 \quad 9 \quad 16 \quad 9 \quad 7 \quad 4 \quad 9 \quad 11$$

without modifying a or b.

■■ **E7.24** Add a method

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

to the Sequence class of Exercise • E7.23 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

$$1 \quad 4 \quad 9 \quad 16$$

and b is

$$9 \quad 7 \quad 4 \quad 9 \quad 11$$

then a.merge(b) returns the sequence

$$1 \quad 9 \quad 4 \quad 7 \quad 9 \quad 4 \quad 16 \quad 9 \quad 11$$

without modifying a or b.

■■ **E7.25** Add a method

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

to the Sequence class of Exercise • E7.23 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

$$1 \quad 4 \quad 9 \quad 16$$

and b is

$$4 \quad 7 \quad 9 \quad 9 \quad 11$$

then a.mergeSorted(b) returns the sequence

$$1 \quad 4 \quad 4 \quad 7 \quad 9 \quad 9 \quad 9 \quad 11 \quad 16$$

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

PROGRAMMING PROJECTS

■■ **P7.1** A smartphone keeps track of the *x*- and *y*-position of a person once per minute. Use this data to solve a vexing problem: Where did the person last park their car? Look for positions that indicate a stopped vehicle, then a person walking out of the car.

■■ **P7.2** 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:

```
inRun = 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).*

∎∎ P7.3 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.

∎∎ P7.4 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.

∎∎∎ P7.5 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.

∎∎∎ P7.6 *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?

■■■ **P7.7** 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.

■■ **P7.8** 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
```


© lepas2004/iStockphoto.

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.

■■■ **P7.9** 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.

© KathyMuller/iStockphoto.

■■■ **P7.10** 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

■■■ **P7.11** *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.

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.
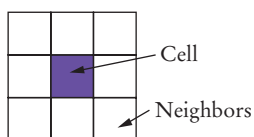

Figure 21
Neighborhood of a Cell

Generation 0   Generation 1   Generation 2   Generation 3   Generation 4
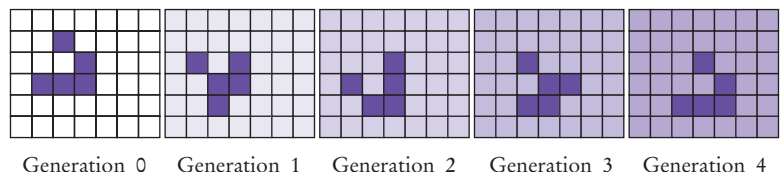
Figure 22   Glider

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.
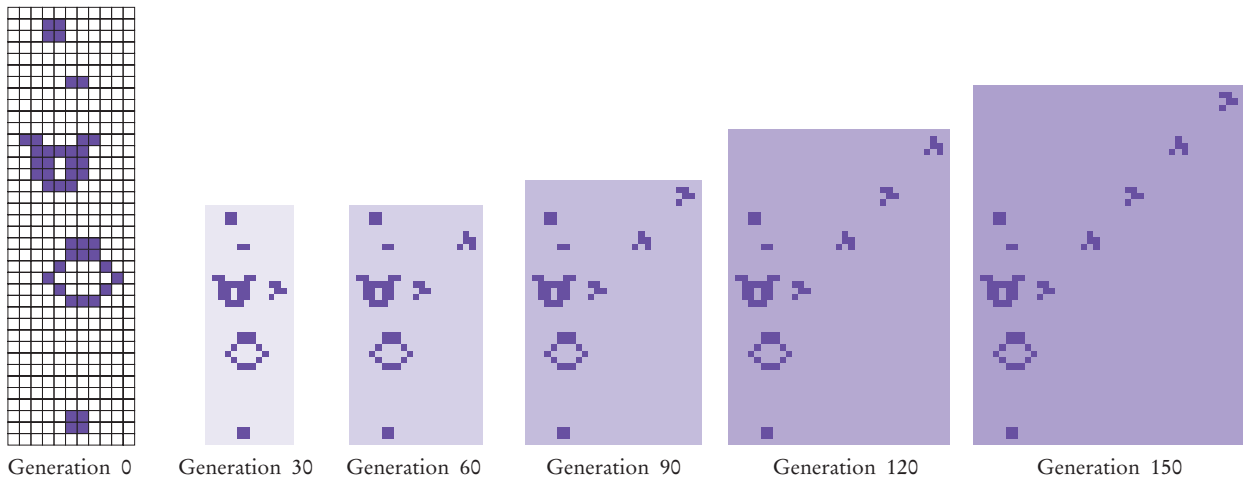

| Generation 0 | Generation 30 | Generation 60 | Generation 90 | Generation 120 | Generation 150 |

Figure 23 Glider Gun

■■ **Business P7.12** 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()
```


© joshblake/iStockphoto.

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 P7.13** 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 P7.14 Improve the program of Exercise ●● Business P7.13 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 P7.15 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.

© GordonHeeley/iStockphoto.

■■■ Science P7.16 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:

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

© nicolamargaret/iStockphoto.

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 P7.17** 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 P7.18** 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 $v\Delta t$.
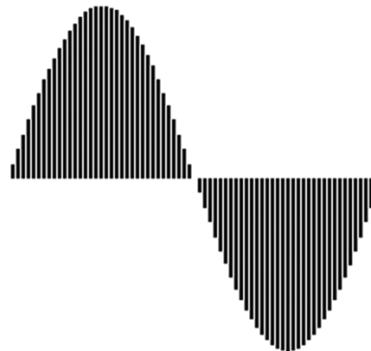
Every ten iterations, plot the spring displacement as a bar, where 1 pixel represents 1 cm, as shown here.
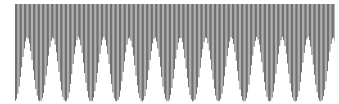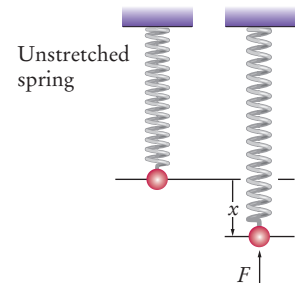
■■ **Graphics P7.19** Generate the image of a checkerboard.
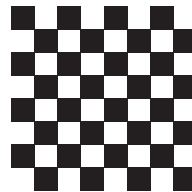
■ **Graphics P7.20** Generate the image of a sine wave. Draw a line of pixels for every five degrees.

Graphics P7.21 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 P7.22 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 P7.23 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 P7.24 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 P7.25 Improve the `BarChart` class of Exercise •• Graphics P7.24 to work correctly when the data contains negative values.

Graphics P7.26 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.