

ARRAYS AND ARRAY LISTS



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

CHAPTER CONTENTS

6.1 ARRAYS 250

Syntax 6.1: Arrays 251

Common Error 6.1: Bounds Errors 255

Common Error 6.2: Uninitialized Arrays 255

Programming Tip 6.1: Use Arrays for Sequences of Related Items 256

Random Fact 6.1: An Early Internet Worm 256

6.2 THE ENHANCED FOR LOOP 257

Syntax 6.2: The Enhanced for Loop 258

6.3 COMMON ARRAY ALGORITHMS 258

Common Error 6.3: Underestimating the Size of a Data Set 267

Special Topic 6.1: Sorting with the Java Library 267

Special Topic 6.2: Binary Search 267

6.4 USING ARRAYS WITH METHODS 268

Special Topic 6.3: Methods with a Variable Number of Parameters 272

6.5 PROBLEM SOLVING: ADAPTING ALGORITHMS 272

Programming Tip 6.2: Reading Exception Reports 274

How To 6.1: Working with Arrays 275

Worked Example 6.1: Rolling the Dice +

6.6 PROBLEM SOLVING: DISCOVERING ALGORITHMS BY MANIPULATING PHYSICAL OBJECTS 279

Video Example 6.1: Removing Duplicates from an Array +

6.7 TWO-DIMENSIONAL ARRAYS 282

Syntax 6.3: Two-Dimensional Array Declaration 283

Worked Example 6.2: A World Population Table +

Special Topic 6.4: Two-Dimensional Arrays with Variable Row Lengths 288

Special Topic 6.5: Multidimensional Arrays 289

6.8 ARRAY LISTS 289

Syntax 6.4: Array Lists 290

Common Error 6.4: Length and Size 299

Special Topic 6.6: The Diamond Syntax in Java 7 299

Video Example 6.2: Game of Life +



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.

6.1 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.

6.1.1 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.

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

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.

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
```

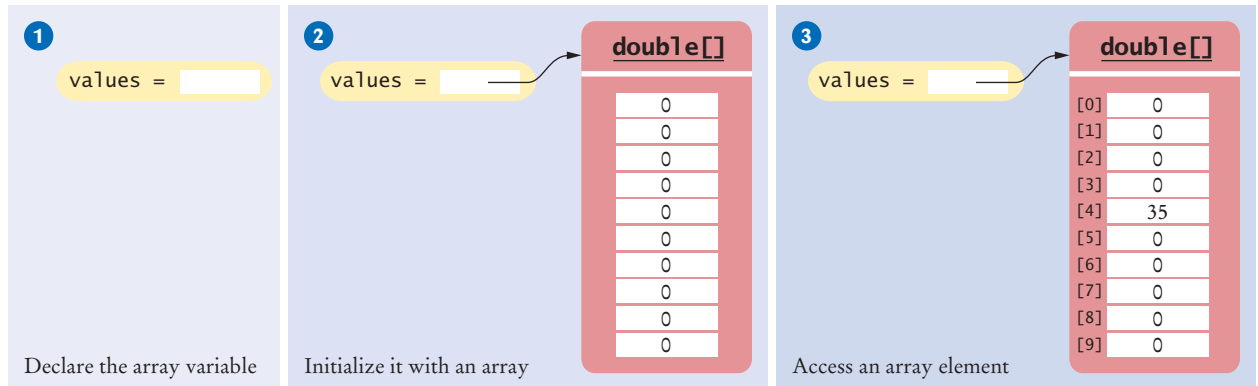


Figure 1 An Array of Size 10

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; ③
```

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 6.1 Arrays

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

To access an element: `arrayReference[index]`

```

Name of array variable      Element type  Length
Type of array variable — double[] values = new double[10];
                           double[] moreValues = { 32, 54, 67.5, 29, 35 };

```

Use brackets to access an element.

```
values[i] = 0;
```

The index must be ≥ 0 and $<$ the length of the array.
See page 255.

List of initial values

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

- `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`.




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

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

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

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 6.3.9.

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]`.

6.1.2 Array References

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.) When you access the elements in an array, you need not be concerned about the fact that Java uses array references. This only becomes important when copying array references.

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

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

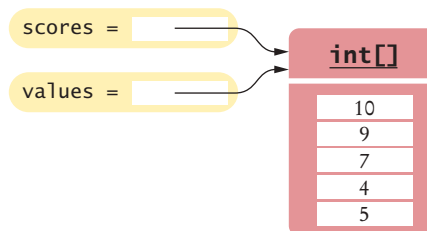


Figure 2
Two Array Variables Referencing the Same Array

6.1.3 Partially Filled Arrays



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:

```
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:

```
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.

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]);
}
```

ONLINE EXAMPLE

⊕ A program demonstrating array operations.

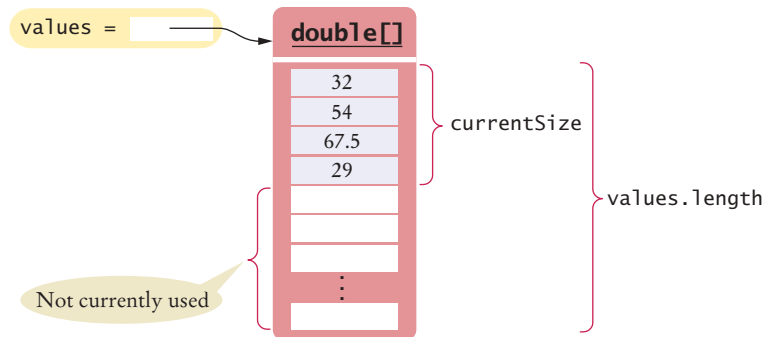


Figure 3 A Partially Filled Array



1. Declare an array of integers containing the first five prime numbers.
2. Assume the array `primes` has been initialized as described in Self Check 1. What does it contain after executing the following loop?


```
for (int i = 0; i < 2; i++)
{
    primes[4 - i] = primes[i];
}
```
3. Assume the array `primes` has been initialized as described in Self Check 1. What does it contain after executing the following loop?


```
for (int i = 0; i < 5; i++)
{
    primes[i]++;
}
```
4. Given the declaration


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

 write statements to put the integer 10 into the elements of the array `values` with the lowest and the highest valid index.
5. Declare an array called `words` that can hold ten elements of type `String`.
6. Declare an array containing two strings, "Yes", and "No".
7. Can you produce the output on page 250 without storing the inputs in an array, by using an algorithm similar to the algorithm for finding the maximum in Section 4.7.5?

Practice It Now you can try these exercises at the end of the chapter: R6.1, R6.2, R6.6, P6.1.

Common Error 6.1



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 6.2



Uninitialized 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
```

The Java compiler will catch this error. The remedy is to initialize the variable with an array:

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

Programming Tip 6.1

**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.

**Random Fact 6.1** An Early Internet Worm

In November 1988, Robert Morris, a student at Cornell University, launched a so-called virus program that infected about 6,000 computers connected to the Internet across the United States. Tens of thousands of computer users were unable to read their e-mail or otherwise use their computers. All major universities and many high-tech companies were affected. (The Internet was much smaller then than it is now.)

The particular kind of virus used in this attack is called a *worm*. The worm program crawled from one computer on the Internet to the next. The worm would attempt to connect to *finger*, a program in the UNIX operating system for finding information on a user who has an account on a particular computer on the network. Like many programs in UNIX, *finger* was written in the C language. In order to store the user name, the *finger* program allocated an array of 512 characters, under the assumption that nobody would ever provide such a long input. Unfortunately, C 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. In some versions of the *finger* program, the programmer had been lazy and had not checked whether the array holding the input characters was large enough

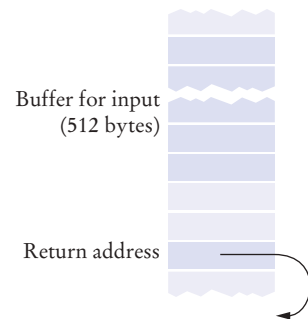
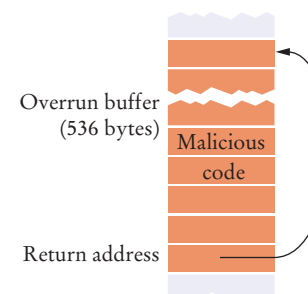
to hold the input. So the worm program purposefully filled the 512-character array with 536 bytes. The excess 24 bytes would overwrite a return address, which the attacker knew was stored just after the array. When that method was finished, it didn't return to its caller but to code supplied by the worm (see the figure, A "Buffer Overrun" Attack). That code ran under the same super-user privileges as *finger*, allowing the worm to gain entry into the remote system. Had the programmer who wrote *finger* been more conscientious, this particular attack would not be possible.

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 many weeks to plan the antisocial act of breaking into thousands of computers and disabling them. 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 3 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 steal financial data or use the attacked computers for sending spam e-mail. Sadly, many of these attacks continue to be possible because of poorly written programs that are susceptible to buffer overrun errors.

1 Before the attack**2** After the attack

A "Buffer Overrun" Attack

6.2 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 and 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:

```
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:

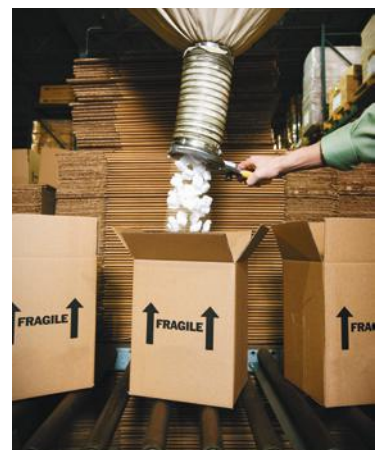
```
for (int i = 0; i < values.length; i++)
{
    values[i] = 0; // OK
}
```

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

ONLINE EXAMPLE

⊕ An program that demonstrates the enhanced for loop.

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



Syntax 6.2 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

These statements
are executed for each
element.

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

The variable
contains an element,
not an index.



8. What does this enhanced for loop do?


```
int counter = 0;
for (double element : values)
{
    if (element == 0) { counter++; }
}
```
9. Write an enhanced for loop that prints all elements in the array `values`.
10. Write an enhanced for loop that multiplies all elements in a `double[]` array named `factors`, accumulating the result in a variable named `product`.
11. Why is the enhanced for loop not an appropriate shortcut for the following basic for loop?


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

Practice It Now you can try these exercises at the end of the chapter: R6.7, R6.8, R6.9.

6.3 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.

6.3.1 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;
}
```

6.3.2 Sum and Average Value

You have already encountered this algorithm in Section 4.7.1. 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; }
```

6.3.3 Maximum and Minimum



Use the algorithm from Section 4.7.5 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];
    }
}
```

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.

6.3.4 Element Separators

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

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
```

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:

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



To print five elements, you need four separators.

If you want comma separators, you can use the `Arrays.toString` method. The expression

```
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:

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

6.3.5 Linear Search



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

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:

```
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—see Special Topic 6.2 on page 267.

6.3.6 Removing an Element

Suppose you want to remove the element with index `pos` from the array `values`. As explained in Section 6.1.3, 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 `currentSize` variable. (See Figure 4.)

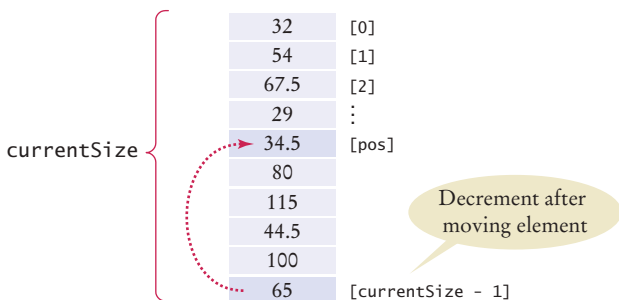


Figure 4
Removing an Element in an Unordered Array

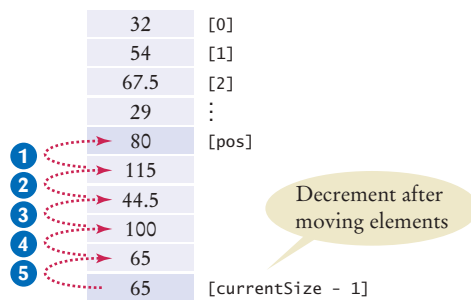


Figure 5
Removing an Element in an Ordered Array

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



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 5.)

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

6.3.7 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 6.1.3.

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;
}
```

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 7).

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.

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

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

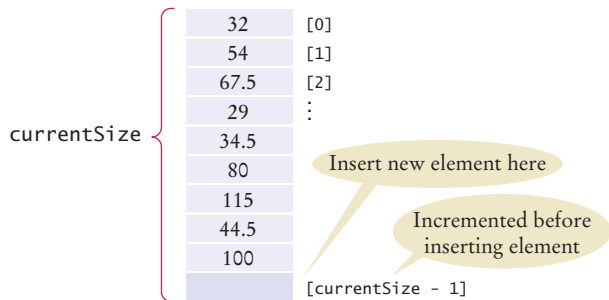


Figure 6
Inserting an Element in an Unordered Array

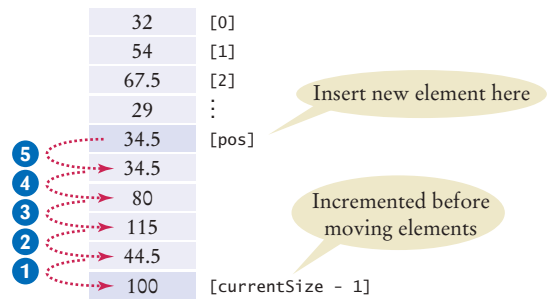


Figure 7
Inserting an Element in an Ordered Array

6.3.8 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.

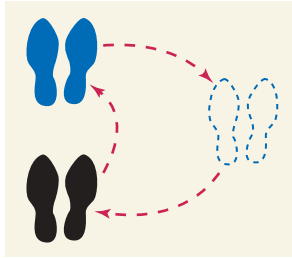
Consider the task of swapping the elements at positions i and j of an array `values`. We'd like to set `values[i]` to `values[j]`. But that overwrites the value that is currently stored in `values[i]`, so we want to save that first:

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

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

```
values[j] = temp;
```

Figure 8 shows the process.



To swap two elements, you need a temporary variable.

Use a temporary variable when swapping two elements.

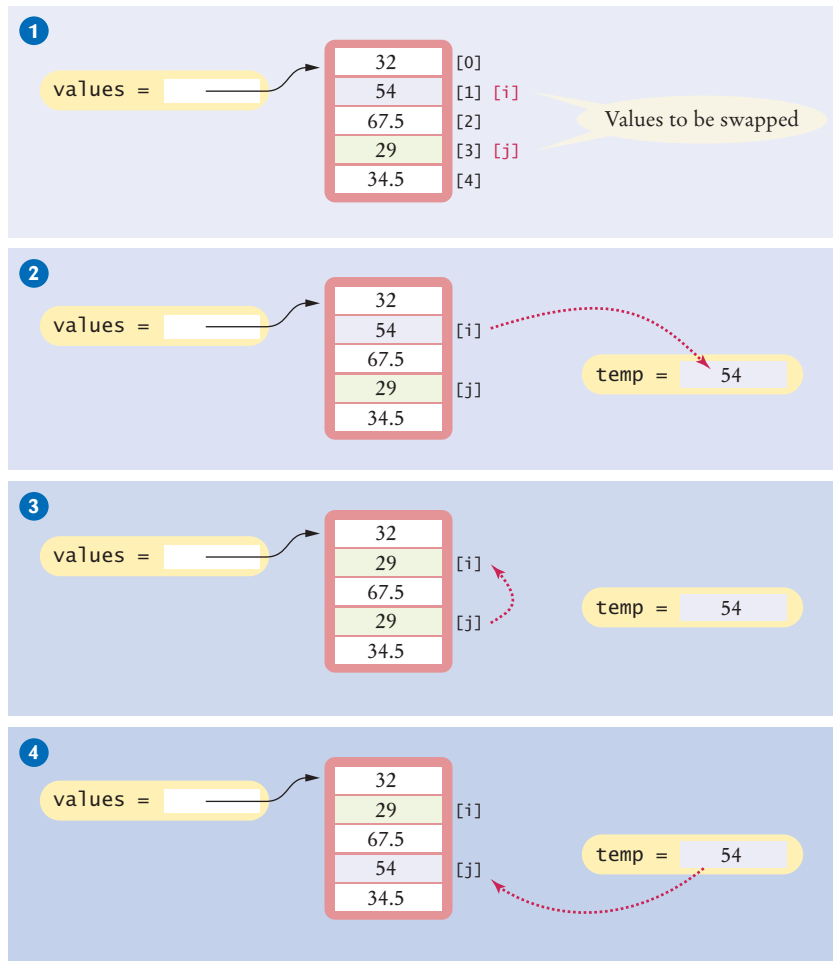


Figure 8 Swapping Array Elements

6.3.9 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 9):

```
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 9).

```
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.

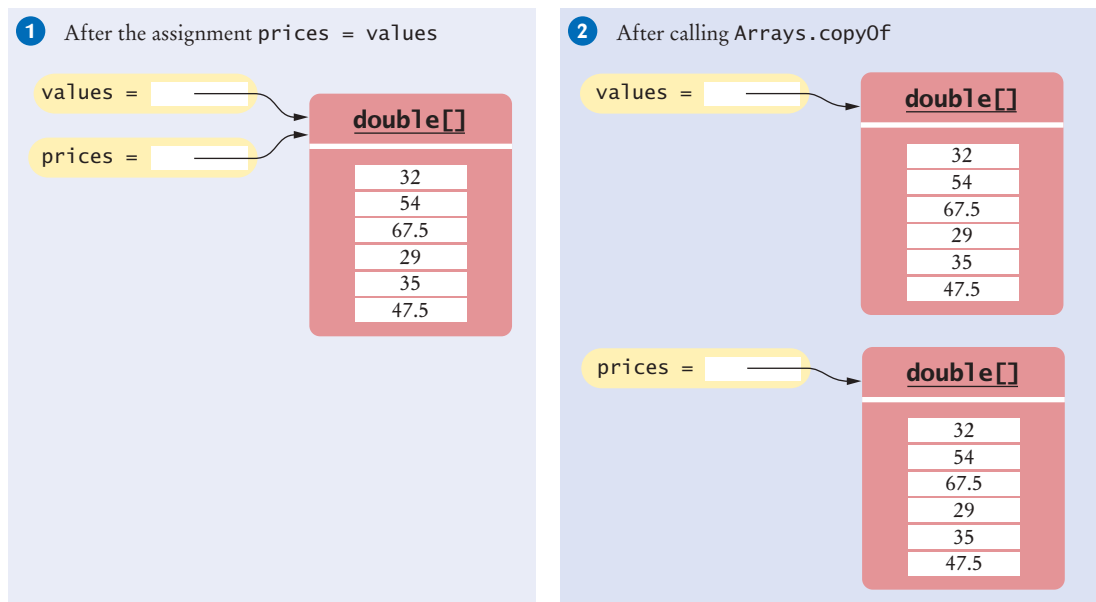


Figure 9 Copying an Array Reference versus Copying an Array

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 10):

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

The `copyOf` method was added in Java 6. If you use Java 5, replace

```
double[] newValues = Arrays.copyOf(values, n)
```

with

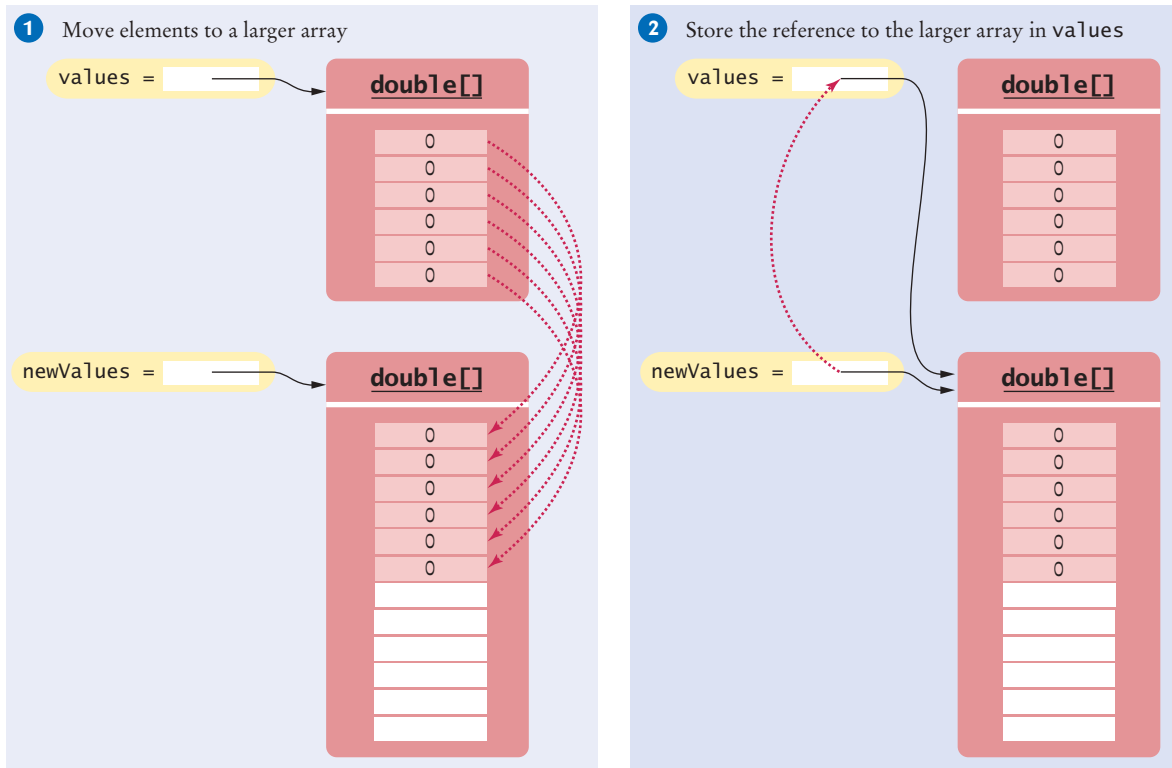


Figure 10 Growing an Array

```
double[] newValues = new double[n];
for (int i = 0; i < n && i < values.length; i++)
{
    newValues[i] = values[i];
}
```

6.3.10 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.

section_3/LargestInArray.java

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

```

34
35     // Print all values, marking the largest
36
37     for (int i = 0; i < currentSize; i++)
38     {
39         System.out.print(values[i]);
40         if (values[i] == largest)
41         {
42             System.out.print(" <== largest value");
43         }
44         System.out.println();
45     }
46 }
47 }

```

Program Run

```

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

```



SELF CHECK

12. Given these inputs, what is the output of the LargestInArray program?

```
20 10 20 Q
```

13. Write a loop that counts how many elements in an array are equal to zero.
14. Consider the algorithm to find the largest element in an array. Why don't we initialize `largest` and `i` with zero, like this?

```

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

```

15. When printing separators, we skipped the separator before the initial element. Rewrite the loop so that the separator is printed *after* each element, except for the last element.

16. What is wrong with these statements for printing an array with separators?

```

System.out.print(values[0]);
for (int i = 1; i < values.length; i++)
{
    System.out.print(", " + values[i]);
}

```

17. When finding the position of a match, we used a `while` loop, not a `for` loop. What is wrong with using this loop instead?

```

for (pos = 0; pos < values.length && !found; pos++)
{
    if (values[pos] > 100)
    {
        found = true;
    }
}

```

}

18. When inserting an element into an array, we moved the elements with larger index values, starting at the end of the array. Why is it wrong to start at the insertion location, like this?

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

Practice It Now you can try these exercises at the end of the chapter: R6.17, R6.20, P6.15.

Common Error 6.3



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 6.1



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);
```



Special Topic 6.2



Binary Search

When an array is sorted, there is a much faster search algorithm than the linear search of Section 6.3.5.

Consider the following sorted array values.

```
[0] [1] [2] [3] [4] [5] [6] [7]
 1  5  8  9 12 17 20 32
```

We would like to see whether the number 15 is in the array. Let's narrow our search by finding whether the number is in the first or second half of the array. The last point in the first half of the `values` array, `values[3]`, is 9, which is smaller than the number we are looking for. Hence, we should look in the second half of the array for a match, that is, in the sequence:

```
[0] [1] [2] [3] [4] [5] [6] [7]
 1  5  8  9 12 17 20 32
```

Now the last element of the first half of this sequence is 17; hence, the number must be located in the sequence:

```
[0] [1] [2] [3] [4] [5] [6] [7]
 1  5  8  9 12 17 20 32
```

The last element of the first half of this very short sequence is 12, which is smaller than the number that we are searching, so we must look in the second half:

```
[0] [1] [2] [3] [4] [5] [6] [7]
 1  5  8  9 12 17 20 32
```

We still don't have a match because $15 \neq 17$, and we cannot divide the subsequence further. If we wanted to insert 15 into the sequence, we would need to insert it just before values[5].

This search process is called a **binary search**, because we cut the size of the search in half in each step. That cutting in half works only because we know that the array is sorted. Here is an implementation in Java:

```
boolean found = false;
int low = 0;
int high = values.length - 1;
int pos = 0;
while (low <= high && !found)
{
    pos = (low + high) / 2; // Midpoint of the subsequence
    if (values[pos] == searchedNumber) { found = true; }
    else if (values[pos] < searchedNumber) { low = pos + 1; } // Look in second half
    else { high = pos - 1; } // Look in first half
}
if (found) { System.out.println("Found at position " + pos); }
else { System.out.println("Not found. Insert before position " + pos); }
```

6.4 Using Arrays with Methods

Arrays can occur as method arguments and return values.

In this section, we will explore how to write methods that process arrays.

When you define a method with an array argument, you provide a parameter variable for the array. For example, the following method computes the sum of an array of floating-point numbers:

```
public static double sum(double[] values)
{
    double total = 0;
    for (double element : values)
    {
        total = total + element;
    }
    return total;
}
```

This method visits the array elements, but it does not modify them. It is also possible to modify the elements of an array. The following method multiplies all elements of an array by a given factor:

```
public static void multiply(double[] values, double factor)
{
    for (int i = 0; i < values.length; i++)
    {
```

```

        values[i] = values[i] * factor;
    }
}

```

Figure 11 traces the method call

```
multiply(scores, 10);
```

Note these steps:

- The parameter variables `values` and `factor` are created. **1**
- The parameter variables are initialized with the arguments that are passed in the call. In our case, `values` is set to `scores` and `factor` is set to 10. Note that `values` and `scores` are references to the *same* array. **2**
- The method multiplies all array elements by 10. **3**
- The method returns. Its parameter variables are removed. However, `scores` still refers to the array with the modified elements. **4**

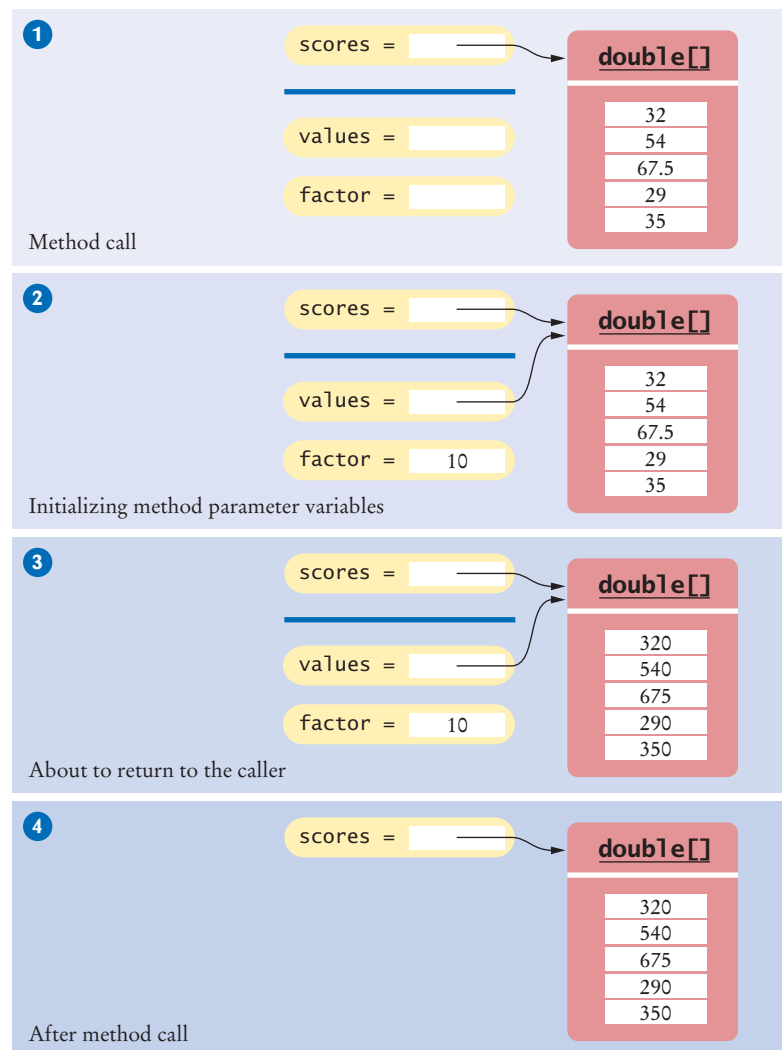


Figure 11
Trace of Call to
the `multiply` Method

A method can return an array. Simply build up the result in the method and return it. In this example, the `squares` method returns an array of squares from 0^2 up to $(n-1)^2$:

```
public static int[] squares(int n)
{
    int[] result = new int[n];
    for (int i = 0; i < n; i++)
    {
        result[i] = i * i;
    }
    return result;
}
```

The following example program reads values from standard input, multiplies them by 10, and prints the result in reverse order. The program uses three methods:

- The `readInputs` method returns an array, using the algorithm of Section 6.3.10.
- The `multiply` method has an array argument. It modifies the array elements.
- The `printReversed` method also has an array argument, but it does not modify the array elements.

section_4/Reverse.java

```
1  import java.util.Scanner;
2
3  /**
4   * This program reads, scales, and reverses a sequence of numbers.
5   */
6  public class Reverse
7  {
8      public static void main(String[] args)
9      {
10         double[] numbers = readInputs(5);
11         multiply(numbers, 10);
12         printReversed(numbers);
13     }
14
15     /**
16      * Reads a sequence of floating-point numbers.
17      * @param numberOfInputs the number of inputs to read
18      * @return an array containing the input values
19      */
20     public static double[] readInputs(int numberOfInputs)
21     {
22         System.out.println("Enter " + numberOfInputs + " numbers: ");
23         Scanner in = new Scanner(System.in);
24         double[] inputs = new double[numberOfInputs];
25         for (int i = 0; i < inputs.length; i++)
26         {
27             inputs[i] = in.nextDouble();
28         }
29         return inputs;
30     }
31
32     /**
33      * Multiplies all elements of an array by a factor.
34      * @param values an array
35      * @param factor the value with which element is multiplied
36      */
```

```

37 public static void multiply(double[] values, double factor)
38 {
39     for (int i = 0; i < values.length; i++)
40     {
41         values[i] = values[i] * factor;
42     }
43 }
44
45 /**
46  Prints an array in reverse order.
47  @param values an array of numbers
48  @return an array that contains the elements of values in reverse order
49  */
50 public static void printReversed(double[] values)
51 {
52     // Traverse the array in reverse order, starting with the last element
53     for (int i = values.length - 1; i >= 0; i--)
54     {
55         System.out.print(values[i] + " ");
56     }
57     System.out.println();
58 }
59 }

```

Program Run

```

Enter 5 numbers:
12 25 20 0 10
100.0 0.0 200.0 250.0 120.0

```



SELF CHECK

19. How do you call the squares method to compute the first five squares and store the result in an array numbers?
20. Write a method fill that fills all elements of an array of integers with a given value. For example, the call fill(scores, 10) should fill all elements of the array scores with the value 10.
21. Describe the purpose of the following method:

```

public static int[] mystery(int length, int n)
{
    int[] result = new int[length];
    for (int i = 0; i < result.length; i++)
    {
        result[i] = (int) (n * Math.random());
    }
    return result;
}

```

22. Consider the following method that reverses an array:

```

public static int[] reverse(int[] values)
{
    int[] result = new int[values.length];
    for (int i = 0; i < values.length; i++)
    {
        result[i] = values[values.length - 1 - i];
    }
    return result;
}

```

Suppose the reverse method is called with an array scores that contains the numbers 1, 4, and 9. What is the contents of scores after the method call?

23. Provide a trace diagram of the reverse method when called with an array that contains the values 1, 4, and 9.

Practice It Now you can try these exercises at the end of the chapter: R6.25, P6.6, P6.7.

Special Topic 6.3



Methods with a Variable Number of Parameters

Starting with Java version 5.0, it is possible to declare methods that receive a variable number of parameters. For example, we can write a sum method that can compute the sum of any number of arguments:

```
int a = sum(1, 3); // Sets a to 4
int b = sum(1, 7, 2, 9); // Sets b to 19
```

The modified sum method must be declared as

```
public static void sum(int... values)
```

The ... symbol 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 sum(int... values)
{
    int total = 0;
    for (int i = 0; i < values.length; i++) // values is an int[]
    {
        total = total + values[i];
    }
    return total;
}
```

6.5 Problem Solving: Adapting Algorithms

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

In Section 6.3, 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.

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 6.3.2)
- Finding the minimum value (Section 6.3.3)
- Removing an element (Section 6.3.6)

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

Find the minimum.
Remove it 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 6.3.6 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 6.3.5)

We need to fix our plan of attack:

Find the minimum value.
Find its position.
Remove that position from the array.
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. 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;
}
```

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

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;
    }
}
```

ONLINE EXAMPLE

- ⊕ A program that computes the final score using the adapted algorithm for finding the minimum.

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

- Find the position of the minimum.
- Remove it from the array.
- 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.

SELF CHECK

24. Section 6.3.6 has two algorithms for removing an element. Which of the two should be used to solve the task described in this section?
25. It isn't actually necessary to *remove* the minimum in order to compute the total score. Describe an alternative.
26. How can you print the number of positive and negative values in a given array, using one or more of the algorithms in Section 4.7?
27. How can you print all positive values in an array, separated by commas?
28. Consider the following algorithm for collecting all matches in an array:

```
int matchesSize = 0;
for (int i = 0; i < values.length; i++)
{
    if (values[i] fulfills the condition)
    {
        matches[matchesSize] = values[i];
        matchesSize++;
    }
}
```

How can this algorithm help you with Self Check 27?

Practice It Now you can try these exercises at the end of the chapter: R6.26, R6.27.

Programming Tip 6.2**Reading Exception Reports**

You will sometimes have programs that terminate, reporting an “exception”, such as

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
    at Homework1.processValues(Homework1.java:14)
    at Homework1.main(Homework1.java:36)
```

Quite a few students give up at that point, saying “it didn’t work”, or “my program died”, without reading the error message. Admittedly, the format of the exception report is not very friendly. But, with some practice, it is easy to decipher it.

There are two pieces of useful information:

1. The name of the exception, such as `ArrayIndexOutOfBoundsException`
2. The stack trace, that is, the method calls that led to the exception, such as `Homework1.java:14` and `Homework1.java:36` in our example.

The name of the exception is always in the first line of the report, and it ends in `Exception`. If you get an `ArrayIndexOutOfBoundsException`, then there was a problem with an invalid array index. That is useful information.

To determine the line number of the offending code, look at the file names and line numbers. The first line of the stack trace is the method that actually generated the exception. The last line of the stack trace is a line in `main`. In our example, the exception was caused by line 14 of `Homework1.java`. Open up the file, go to that line, and look at it! Also look at the name of the exception. In most cases, these two pieces of information will make it completely obvious what went wrong, and you can easily fix your error.

Sometimes, the exception was thrown by a method that is in the standard library. Here is a typical example:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String index
out of range: -4
    at java.lang.String.substring(String.java:1444)
    at Homework2.main(Homework2.java:29)
```

The exception happened in the `substring` method of the `String` class, but the real culprit is the first method in a file that you wrote. In this example, that is `Homework2.main`, and you should look at line 29 of `Homework2.java`.

HOW TO 6.1



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.

Consider again the problem from Section 6.5: 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.



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 6.3. 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 6.3. That is the case with calculating the sum (Section 6.3.2) and reading the inputs (Section 6.3.10). At other times, you need to combine several algorithms. To remove the minimum value, you can find the minimum value (Section 6.3.3), find its position (Section 6.3.5), and remove the element at that position (Section 6.3.6).

We have now refined our plan as follows:

Read inputs.
Find the minimum.
Find its position.
Remove the minimum.
Calculate the sum.

This plan will work—see Section 6.5. 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.

Step 3 Use 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 make each processing step into a separate method. In our example, we will implement three methods:

- `readInputs`
- `sum`
- `minimum`

The `main` method simply calls these methods:

```
double[] scores = readInputs();
double total = sum(scores) - minimum(scores);
System.out.println("Final score: " + total);
```

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. |

Here's the complete program (`how_to_1/Scores.java`):

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

/**
 * This program computes a final score for a series of quiz scores: the sum after dropping
 * the lowest score. The program uses arrays.
 */
public class Scores
{
    public static void main(String[] args)
    {
        double[] scores = readInputs();
        if (scores.length == 0)
        {
            System.out.println("At least one score is required.");
        }
        else
        {
            double total = sum(scores) - minimum(scores);
            System.out.println("Final score: " + total);
        }
    }

    /**
     * Reads a sequence of floating-point numbers.
     * @return an array containing the numbers
     */
    public static double[] readInputs()
    {
        // Read the input values into an array

        final int INITIAL_SIZE = 10;
        double[] inputs = new double[INITIAL_SIZE];
        System.out.println("Please enter values, Q to quit:");
        Scanner in = new Scanner(System.in);
        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);
        }
        inputs[currentSize] = in.nextDouble();
        currentSize++;
    }

    return Arrays.copyOf(inputs, currentSize);
}

/**
 * Computes the sum of the values in an array.
 * @param values an array
 * @return the sum of the values in values
 */
public static double sum(double[] values)
{
    double total = 0;
    for (double element : values)
    {
        total = total + element;
    }
    return total;
}

/**
 * Gets the minimum value from an array.
 * @param values an array of size >= 1
 * @return the smallest element of values
 */
public static double minimum(double[] values)
{
    double smallest = values[0];
    for (int i = 1; i < values.length; i++)
    {
        if (values[i] < smallest)
        {
            smallest = values[i];
        }
    }
    return smallest;
}
}

```

WORKED EXAMPLE 6.1**Rolling the Dice**

This Worked Example shows how to analyze a set of die tosses to see whether the die is “fair”.



6.6 Problem Solving: Discovering Algorithms by Manipulating Physical Objects



Manipulating physical objects can give you ideas for discovering algorithms.

In Section 6.5, 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 6.3.7) or swapped (Section 6.3.8), 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:



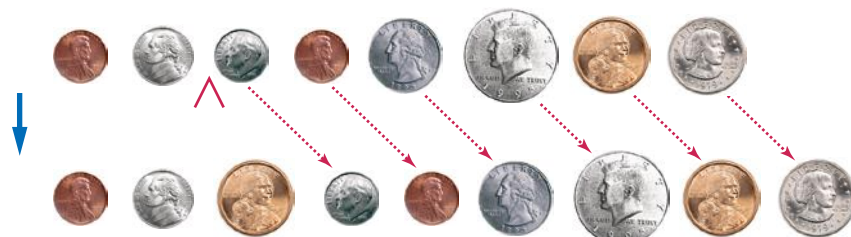
Now let's step back and see what we can do to change the order of the coins. We can remove a coin (Section 6.3.6):

Visualizing the removal of an array element



We can insert a coin (Section 6.3.7):

Visualizing the insertion of an array element



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

Or we can swap two coins (Section 6.3.8).

Visualizing the swapping of two coins



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 $\text{size} / 2$.

And how many iterations do we make? We need to swap all coins in the first half. That is, we need to swap $\text{size} / 2$ coins. The pseudocode is

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

ONLINE EXAMPLE

⊕ A program that implements the algorithm that switches the first and second halves of an array.

You can use paper clips as position markers or counters.

It is a good idea to make a walkthrough of the pseudocode (see Section 4.2). 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. Self Check 29 asks you to carry out the walkthrough, and Exercise P6.8 asks you to translate the pseudocode to Java. Exercise R6.28 suggests a different algorithm for switching the two halves of an array, by repeatedly removing and inserting coins.

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!



29. Walk through the algorithm that we developed in this section, using two paper clips to indicate the positions for i and j . Explain why there are no bounds errors in the pseudocode.
30. Take out some coins and simulate the following pseudocode, using two paper clips to indicate the positions for i and j .

```
i = 0
j = size - 1
While (i < j)
    Swap elements at positions i and j
    i++
    j--
```

What does the algorithm do?

31. Consider the task of rearranging all elements in an array so that the even numbers come first. Otherwise, the order doesn't matter. For example, the array

1 4 14 2 1 3 5 6 23

could be rearranged to

4 2 14 6 1 5 3 23 1

Using coins and paperclips, discover an algorithm that solves this task by swapping elements, then describe it in pseudocode.

- 32. Discover an algorithm for the task of Self Check 31 that uses removal and insertion of elements instead of swapping.
- 33. Consider the algorithm in Section 4.7.4 that finds the largest element in a sequence of inputs — *not* the largest element in an array. Why is this algorithm better visualized by picking playing cards from a deck rather than arranging toy soldiers in a sequence?



Practice It Now you can try these exercises at the end of the chapter: R6.28, R6.29, P6.8.

VIDEO EXAMPLE 6.1

Removing Duplicates from an Array



In this Video Example, we will discover an algorithm for removing duplicates from an array.

6.7 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 12: the medal counts of the figure skating competitions at the 2010 Winter Olympics.



| | Gold | Silver | Bronze |
|---------------|------|--------|--------|
| Canada | 1 | 0 | 1 |
| China | 1 | 1 | 0 |
| Germany | 0 | 0 | 1 |
| Korea | 1 | 0 | 0 |
| Japan | 0 | 1 | 1 |
| Russia | 0 | 1 | 1 |
| United States | 1 | 1 | 0 |

Figure 12 Figure Skating Medal Counts

6.7.1 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:

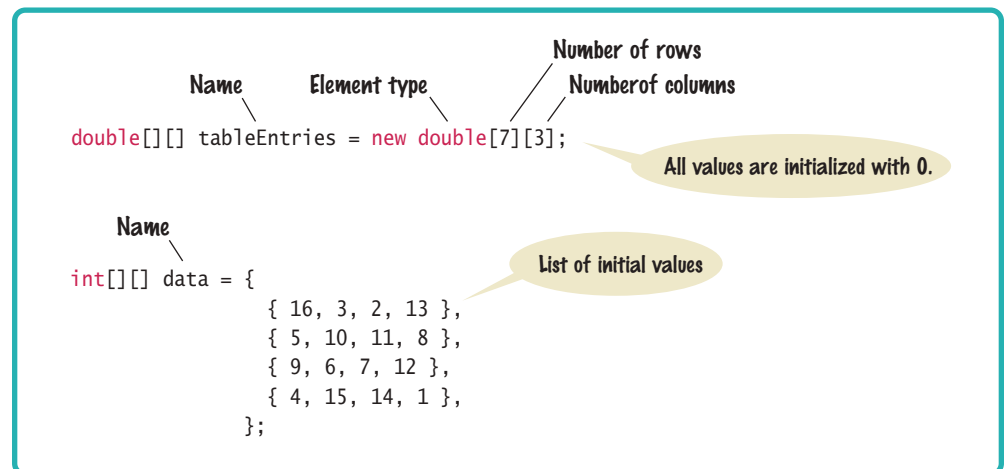
```
final int COUNTRIES = 7;
final int MEDALS = 3;
int[][] counts = new int[COUNTRIES][MEDALS];
```

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

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

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

Syntax 6.3 Two-Dimensional Array Declaration



6.7.2 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 13):

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

To access all elements in a two-dimensional array, you use two 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
}
```

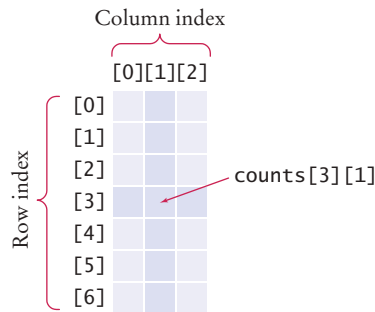


Figure 13
Accessing an Element in a
Two-Dimensional Array

6.7.3 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 14 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:

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

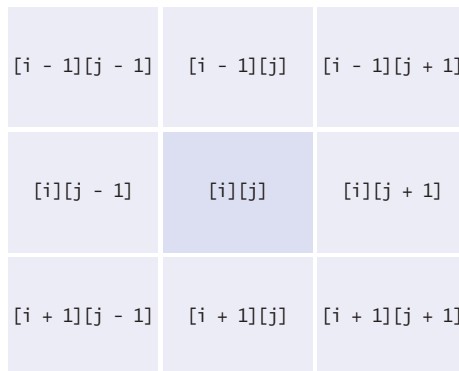
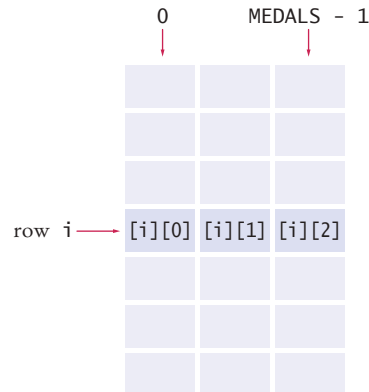


Figure 14
Neighboring Locations in a
Two-Dimensional Array

6.7.4 Computing Row and Column Totals

A common task is to compute row or column totals. In our example, the row totals give us the total number of medals won by a particular country.

Finding the right 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:



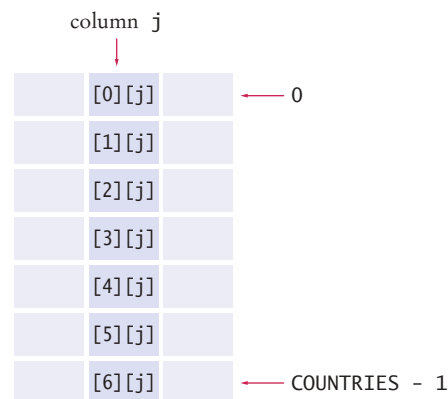
As you can see, we need to compute the sum of `counts[i][j]`, where j ranges from 0 to `MEDALS - 1`. The following loop computes the total:

```
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];
}
```



6.7.5 Two-Dimensional Array Parameters

When you pass a two-dimensional array to a method, you will want to recover 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 6.4 for an explanation of this expression.)

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

```
public static int sum(int[][] values)
{
    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.

section_7/Medals.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 = 7;
9          final int MEDALS = 3;
10
11         String[] countries =
12             {
13                 "Canada",
14                 "China",
15                 "Germany",
16                 "Korea",
17                 "Japan",
18                 "Russia",
19                 "United States"
20             };
21
22         int[][] counts =
23             {
24                 { 1, 0, 1 },
25                 { 1, 1, 0 },
26                 { 0, 0, 1 },
27                 { 1, 0, 0 },
28                 { 0, 1, 1 },
29                 { 0, 1, 1 },
30                 { 1, 1, 0 }
```

```

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

```

Program Run

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



- 34.** What results do you get if you total the columns in our sample data?
- 35.** Consider an 8×8 array for a board game:
- ```
int[][] board = new int[8][8];
```
- Using two nested loops, initialize the board so that zeroes and ones alternate, as on a checkerboard:
- ```

0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
. . .
1 0 1 0 1 0 1 0

```
- Hint:* Check whether $i + j$ is even.
- 36.** Declare a two-dimensional array for representing a tic-tac-toe board. The board has three rows and columns and contains strings "x", "o", and " ".
- 37.** Write an assignment statement to place an "x" in the upper-right corner of the tic-tac-toe board in Self Check 36.
- 38.** Which elements are on the diagonal joining the upper-left and the lower-right corners of the tic-tac-toe board in Self Check 36?

Practice It Now you can try these exercises at the end of the chapter: R6.30, P6.18, P6.19.

WORKED EXAMPLE 6.2

A World Population Table



This Worked Example shows how to print world population data in a table with row and column headers, and with totals for each of the data columns.

Special Topic 6.4



Two-Dimensional Arrays with Variable Row Lengths

When you declare a two-dimensional array with the command

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

then 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 15):

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

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

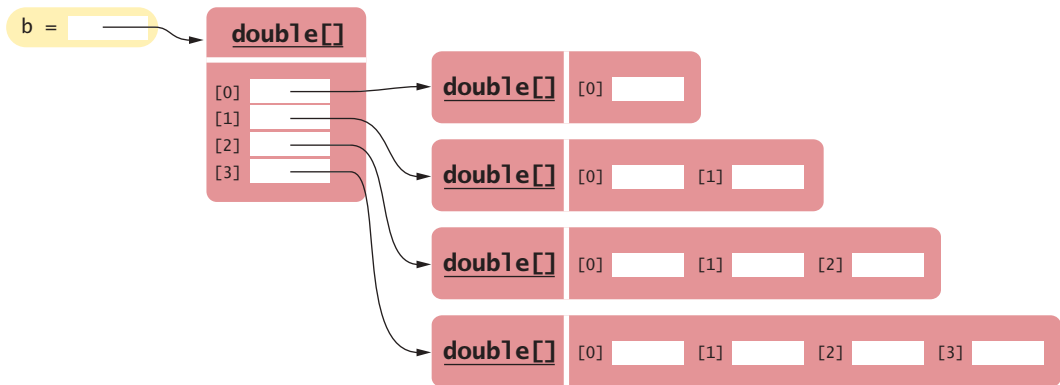


Figure 15 A Triangular Array

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

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

Alternatively, you can use two enhanced for loops:

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

Naturally, such “ragged” arrays are not very common.

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

Special Topic 6.5



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]
```

6.8 Array Lists

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

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

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

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

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



Syntax 6.4 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 $< \text{friends.size}()$.

6.8.1 Declaring and Using Array Lists

The following statement declares an array list of strings:

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

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

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

The type `ArrayList<String>` denotes an array list of `String` elements. The angle brackets around the `String` type tell you that `String` is a **type parameter**. You can replace `String` with any other class and get a different array list type. For that reason, `ArrayList` is called a **generic class**. However, you cannot use primitive types as type parameters—there is no `ArrayList<int>` or `ArrayList<double>`. Section 6.8.5 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. We will discuss constructors in Chapter 8.

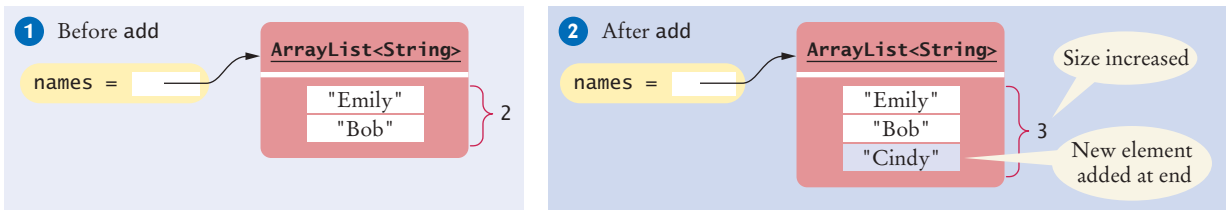


Figure 16 Adding an Element with `add`

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

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

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

The size increases after each call to `add` (see Figure 16). 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:

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

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

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 17).



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

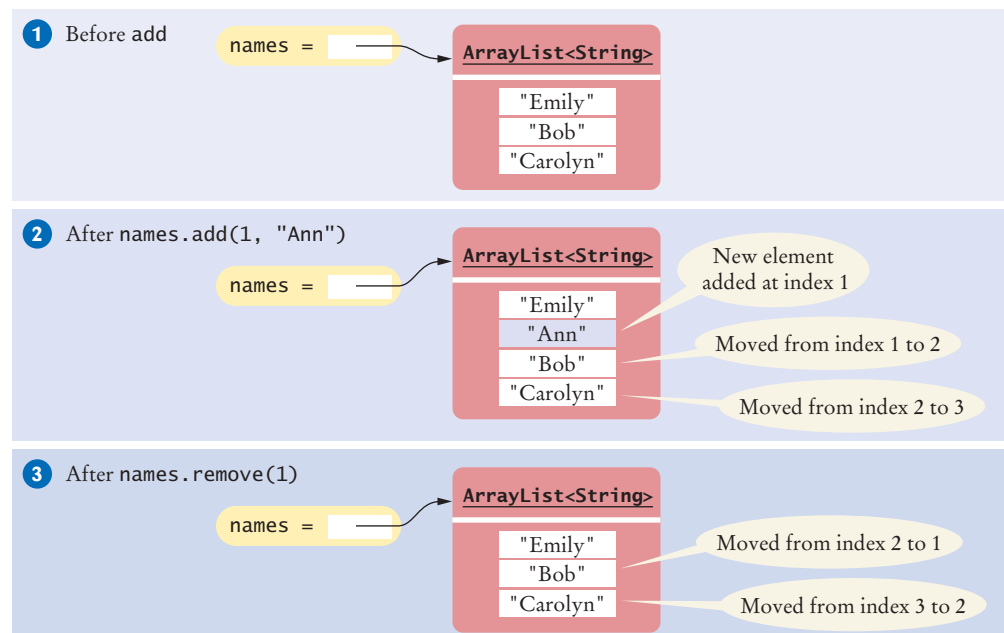


Figure 17
Adding and
Removing
Elements in the
Middle of an
Array List

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 17 illustrates the result of `names.remove(1)`.

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

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

6.8.2 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

| | |
|---|--|
| <code>ArrayList<String> names = new ArrayList<String>();</code> | Constructs an empty array list that can hold strings. |
| <code>names.add("Ann");</code> <code>names.add("Cindy");</code> | Adds elements to the end. |
| <code>System.out.println(names);</code> | Prints [Ann, Cindy]. |
| <code>names.add(1, "Bob");</code> | Inserts an element at index 1. names is now [Ann, Bob, Cindy]. |
| <code>names.remove(0);</code> | Removes the element at index 0. names is now [Bob, Cindy]. |
| <code>names.set(0, "Bill");</code> | Replaces an element with a different value. names is now [Bill, Cindy]. |
| <code>String name = names.get(i);</code> | Gets an element. |
| <code>String last = names.get(names.size() - 1);</code> | Gets the last element. |
| <code>ArrayList<Integer> squares = new ArrayList<Integer>();</code> <code>for (int i = 0; i < 10; i++)</code> <code>{</code> <code> squares.add(i * i);</code> <code>}</code> | Constructs an array list holding the first ten squares. |

6.8.3 Copying Array Lists

As with arrays, you need to remember that array list variables hold references. Copying the reference yields two references to the same array list (see Figure 18).

```
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);
```

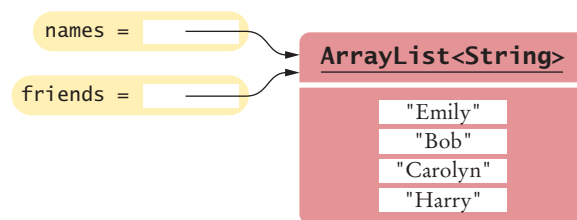


Figure 18 Copying an Array List Reference

6.8.4 Array Lists and Methods

Like arrays, array lists can be method arguments and return values. Here is an example: a method that receives a list of strings and returns the reversed list.

```
public static ArrayList<String> reverse(ArrayList<String> names)
{
    // Allocate a list to hold the method result
    ArrayList<String> result = new ArrayList<String>();

    // Traverse the names list in reverse order, starting with the last element
    for (int i = names.size() - 1; i >= 0; i--)
    {
        // Add each name to the result
        result.add(names.get(i));
    }
    return result;
}
```

If this method is called with an array list containing the names Emily, Bob, Cindy, it returns a new array list with the names Cindy, Bob, Emily.

6.8.5 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).

For example, if you assign a `double` value to a `Double` variable, the number is automatically “put into a box” (see Figure 19).

```
Double wrapper = 29.95;
```

Conversely, wrapper values are automatically “unboxed” to primitive types.

```
double x = wrapper;
```

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

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

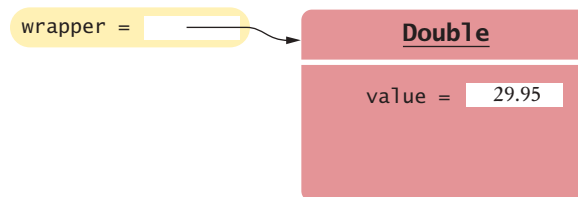


Figure 19 A Wrapper Class Variable

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.

6.8.6 Using Array Algorithms with Array Lists

The array algorithms in Section 6.3 can be converted to array lists simply by using the array list methods instead of the array syntax (see Table 3 on page 297). 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);
    }
}
```

6.8.7 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());
}
```

6.8.8 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.

| <code>i</code> | <code>words</code> |
|----------------|-----------------------------------|
| 0 | "Welcome", "to", "the", "island!" |
| 1 | "Welcome", "the", "island!" |
| 2 | |

We should not increment the index when removing a word. The appropriate pseudo-code 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++;
    }
}

```

6.8.9 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.

The following program shows how to mark the largest value in a sequence of values. This program uses an array list. Note how the program is an improvement over the array version on page 265. This program can process input sequences of arbitrary length.

ONLINE EXAMPLE

⊕ A version of the Scores program using an array list.

Table 3 Comparing Array and Array List Operations

| Operation | Arrays | Array Lists |
|---|---|--|
| Get an element. | <code>x = values[4];</code> | <code>x = values.get(4)</code> |
| Replace an element. | <code>values[4] = 35;</code> | <code>values.set(4, 35);</code> |
| Number of elements. | <code>values.length</code> | <code>values.size()</code> |
| Number of filled elements. | <code>currentSize</code> (companion variable, see Section 6.1.3) | <code>values.size()</code> |
| Remove an element. | See Section 6.3.6 | <code>values.remove(4);</code> |
| Add an element, growing the collection. | See Section 6.3.7 | <code>values.add(35);</code> |
| Initializing a collection. | <code>int[] values = { 1, 4, 9 };</code> | No initializer list syntax; call <code>add</code> three times. |

section_8/LargestInArrayList.java

```

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

```

```

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

```

Program Run

```

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

```



SELF CHECK

39. Declare an array list `primes` of integers that contains the first five prime numbers (2, 3, 5, 7, and 11).
40. Given the array list `primes` declared in Self Check 39, write a loop to print its elements in reverse order, starting with the last element.
41. What does the array list `names` contain after the following statements?

```

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

```

42. What is wrong with this code snippet?

```

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

```

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

```

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

```

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

```

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

```

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

45. The `section_8` directory of your source code contains an alternate implementation of the problem solution in How To 6.1 on page 275. Compare the array and array list implementations. What is the primary advantage of the latter?

Practice It Now you can try these exercises at the end of the chapter: R6.10, R6.34, P6.21, P6.23.

Common Error 6.4



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.

| Data Type | Number of Elements |
|------------|-------------------------|
| Array | <code>a.length</code> |
| Array list | <code>a.size()</code> |
| String | <code>a.length()</code> |

It is a common error to confuse these. You just have to remember the correct syntax for every data type.

Special Topic 6.6



The Diamond Syntax in Java 7

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

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

instead of

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

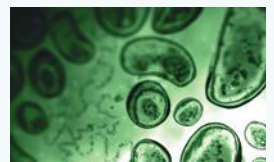
This shortcut is called the “diamond syntax” because the empty brackets `<>` look like a diamond shape.

VIDEO EXAMPLE 6.2



Game of Life

Conway’s *Game of Life* simulates the growth of a population, using only two simple rules. This Video Example shows you how to implement this famous “game”.



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

**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.

Implement methods that process arrays.

- Arrays can occur as method arguments and return values.

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.



STANDARD LIBRARY ITEMS INTRODUCED IN THIS CHAPTER

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

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

REVIEW EXERCISES

- ■ **R6.1** Write code that fills an array values with each set of numbers below.

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

- ■ R6.2 Consider the following array:

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

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

- a.

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

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

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

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

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

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

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

```
int total = 0;
for (int i = 0; i < 10; i++) { total = a[i] - total; }
```
- ■ R6.3 Consider the following array:

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

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

- a.

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

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

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

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

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

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

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

```
for (int i = 1; i < 5; i++) { a[i] = a[9 - i]; }
```
- ■ ■ R6.4 Write a loop that fills an array values with ten random numbers between 1 and 100. Write code for two nested loops that fill values with ten *different* random numbers between 1 and 100.
 - ■ R6.5 Write Java code for a loop that simultaneously computes both the maximum and minimum of an array.
 - R6.6 What is wrong with each of the following code segments?

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

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

- ■ **R6.7** Write enhanced for loops for the following tasks.
 - a. Printing all elements of an array in a single row, separated by spaces.
 - b. Computing the product of all elements in an array.
 - c. Counting how many elements in an array are negative.

- ■ **R6.8** Rewrite the following loops without using the enhanced for loop construct. Here, `values` is an array of floating-point numbers.
 - a. `for (double x : values) { total = total + x; }`
 - b. `for (double x : values) { if (x == target) { return true; } }`
 - c. `int i = 0;`
`for (double x : values) { values[i] = 2 * x; i++; }`

- ■ **R6.9** Rewrite the following loops, using the enhanced for loop construct. Here, `values` is an array of floating-point numbers.
 - a. `for (int i = 0; i < values.length; i++) { total = total + values[i]; }`
 - b. `for (int i = 1; i < values.length; i++) { total = total + values[i]; }`
 - c. `for (int i = 0; i < values.length; i++)`
`{`
`if (values[i] == target) { return i; }`
`}`

- **R6.10** What is wrong with each of the following code segments?
 - a. `ArrayList<int> values = new ArrayList<int>();`
 - b. `ArrayList<Integer> values = new ArrayList();`
 - c. `ArrayList<Integer> values = new ArrayList<Integer>;`
 - d. `ArrayList<Integer> values = new ArrayList<Integer>();`
`for (int i = 1; i <= 10; i++)`
`{`
`values.set(i - 1, i * i);`
`}`
 - e. `ArrayList<Integer> values;`
`for (int i = 1; i <= 10; i++)`
`{`
`values.add(i * i);`
`}`

- **R6.11** What is an index of an array? What are the legal index values? What is a bounds error?

- **R6.12** Write a program that contains a bounds error. Run the program. What happens on your computer?

- **R6.13** Write a loop that reads ten numbers and a second loop that displays them in the opposite order from which they were entered.

- **R6.14** Trace the flow of the linear search loop in Section 6.3.5, where `values` contains the elements 80 90 100 120 110. Show two columns, for `pos` and `found`. Repeat the trace when `values` contains 80 90 100 70.

- **R6.15** Trace both mechanisms for removing an element described in Section 6.3.6. Use an array `values` with elements 110 90 100 120 80, and remove the element at index 2.

- ■ **R6.16** 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.

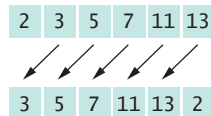
- **R6.17** Trace the flow of the loop in Section 6.3.4 with the given example. Show two columns, one with the value of `i` and one with the output.

- **R6.18** 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`.

- **R6.19** Trace the flow of the loop in Section 6.3.5, 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.
- ■ **R6.20** Trace the algorithm for removing an element described in Section 6.3.6. Use an array `values` with elements 110 90 100 120 80, and remove the element at index 2.
- ■ **R6.21** 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:



- ■ **R6.22** Give pseudocode for an algorithm that removes all negative values from an array, preserving the order of the remaining elements.
- ■ **R6.23** Suppose `values` is a *sorted* array of integers. Give pseudocode that describes how a new value can be inserted in its proper position so that the resulting array stays sorted.
- ■ ■ **R6.24** 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

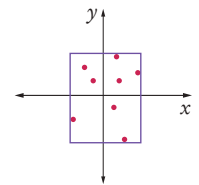
1 2 5 5 3 1 2 4 3 2 2 2 2 3 6 5 5 6 3 1

has length 4.

- **R6.25** What is wrong with the following method that aims to fill an array with random numbers?

```
public static void fillWithRandomNumbers(double[] values)
{
    double[] numbers = new double[values.length];
    for (int i = 0; i < numbers.length; i++)
    {
        numbers[i] = Math.random();
    }
    values = numbers;
}
```

- **R6.26** You are given two arrays denoting x - and y -coordinates of a set of points in the plane. For plotting the point set, we need to know the x - and y -coordinates of the smallest rectangle containing the points.
How can you obtain these values from the fundamental algorithms in Section 6.3?



- **R6.27** Solve the problem described in Section 6.5 by sorting the array first. How do you need to modify the algorithm for computing the total?
- **R6.28** Solve the task described in Section 6.6 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 6.6.
- **R6.29** 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.
- **R6.30** Write Java statements for performing the following tasks with an array declared as


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

 - Fill all entries with 0.
 - Fill elements alternately with 0s and 1s in a checkerboard pattern.
 - Fill only the elements at the top and bottom row with zeroes.
 - Compute the sum of all elements.
 - Print the array in tabular form.
- **R6.31** Write pseudocode for an algorithm that fills the first and last column as well as the first and last row of a two-dimensional array of integers with -1 .
- **R6.32** Section 6.8.8 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.

- ■ **R6.33** 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.

- ■ **R6.34** 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.

- **R6.35** 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.

PROGRAMMING EXERCISES

- ■ **P6.1** Write a program that initializes an array with ten random integers and then prints four lines of output, containing
 - Every element at an even index.
 - Every even element.
 - All elements in reverse order.
 - Only the first and last element.

- ■ **P6.2** Write array methods that carry out the following tasks for an array of integers. For each method, provide a test program.
 - a. Swap the first and last elements in the array.
 - b. Shift all elements by one to the right and move the last element into the first position. For example, 1 4 9 16 25 would be transformed into 25 1 4 9 16.
 - c. Replace all even elements with 0.
 - d. Replace each element except the first and last by the larger of its two neighbors.

- e. Remove the middle element if the array length is odd, or the middle two elements if the length is even.
 - f. Move all even elements to the front, otherwise preserving the order of the elements.
 - g. Return the second-largest element in the array.
 - h. Return true if the array is currently sorted in increasing order.
 - i. Return true if the array contains two adjacent duplicate elements.
 - j. Return true if the array contains duplicate elements (which need not be adjacent).
- **P6.3** Modify the `LargestInArray.java` program in Section 6.3 to mark both the smallest and the largest elements.
 - **P6.4** 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.
 - **P6.5** Write a method `public static void removeMin` that removes the minimum value from a partially filled array without calling other methods.
 - **P6.6** Compute the *alternating sum* of all elements in an array. For example, if your program reads the input

1 4 9 16 9 7 4 9 11

then it computes

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

- **P6.7** Write a method that reverses the sequence of elements in an array. For example, if you call the method with the array

1 4 9 16 9 7 4 9 11

then the array is changed to

11 9 4 7 9 16 9 4 1

- **P6.8** Write a method that implements the algorithm developed in Section 6.6.
- **P6.9** Write a method


```
public static boolean equals(int[] a, int[] b)
```

 that checks whether two arrays have the same elements in the same order.
- **P6.10** Write a method


```
public static boolean sameSet(int[] a, int[] b)
```

 that checks whether two arrays have the same elements in some order, ignoring duplicates. For example, the two arrays

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.

■■■ P6.11 Write a method

```
public static boolean sameElements(int[] a, int[] b)
```

that checks whether two arrays have the same elements in some order, with the same multiplicities. For example,

```
1 4 9 16 9 7 4 9 11
```

and

```
11 1 4 9 16 9 7 4 9
```

would be considered identical, but

```
1 4 9 16 9 7 4 9 11
```

and

```
11 11 7 9 16 4 1 4 9
```

would not. You will probably need one or more helper methods.

- P6.12 A *run* is a sequence of adjacent repeated values. Write a program that generates a sequence of 20 random die tosses in an array and that prints the die values, marking the runs by including them in parentheses, like this:

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

Use the following pseudocode:

Set a boolean variable `inRun` to false.

For each valid index `i` in the array

 If `inRun`

 If `values[i]` is different from the preceding value

 Print `).`

`inRun = false.`

 If not `inRun`

 If `values[i]` is the same as the following value

 Print `(.`

`inRun = true.`

 Print `values[i].`

 If `inRun`, print `).`

- P6.13 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.

- P6.14 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.
- P6.15 Write a program that produces ten random permutations of the numbers 1 to 10. To generate a random permutation, you need to fill an array with the numbers 1 to 10 so that no two entries of the array have the same contents. You could do it by brute force, by generating random values until you have a value that is not yet in the array. But that is inefficient. Instead, follow this algorithm.

Make a second array and fill it with the numbers 1 to 10.

Repeat 10 times

Pick a random element from the second array.

Remove it and append it to the permutation array.

- ■ P6.16 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.

- ■ ■ P6.17 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.

- ■ ■ P6.18 *Magic squares*. An $n \times n$ matrix that is filled with the numbers $1, 2, 3, \dots, n^2$ 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?

- P6.19** 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.

- P6.20** Write a method that computes the average of the neighbors of a two-dimensional array element in the eight directions shown in Figure 14.

```
public static double neighborAverage(int[][] values, int row, int column)
```

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

- P6.21** 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.

- P6.22** Improve the program of Exercise P6.21 to work correctly when the data set contains negative values.

- P6.23** Improve the program of Exercise P6.21 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 *****
    
```

- ■ P6.24 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 40 30
    
```



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

- ■ ■ P6.25 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.



- P6.26 Write a method

```
public static ArrayList<Integer> append(ArrayList<Integer> a, ArrayList<Integer> b)
```

that appends one array list after another. For example, if a is

```
1 4 9 16
```

and b is

```
9 7 4 9 11
```

then `append` returns the array list

```
1 4 9 16 9 7 4 9 11
```

- ■ P6.27 Write a method

```
public static ArrayList<Integer> merge(ArrayList<Integer> a, ArrayList<Integer> b)
```

that merges two array lists, alternating elements from both array lists. If one array list is shorter than the other, then alternate as long as you can and then append the remaining elements from the longer array list. For example, if a is

```
1 4 9 16
```

and b is

```
9 7 4 9 11
```

then `merge` returns the array list

```
1 9 4 7 9 4 16 9 11
```

■ ■ P6.28 Write a method

```
public static ArrayList<Integer> mergeSorted(ArrayList<Integer> a,
      ArrayList<Integer> b)
```

that merges two *sorted* array lists, producing a new sorted array list. Keep an index into each array list, indicating how much of it has been processed already. Each time, append the smallest unprocessed element from either array list, then advance the index. For example, if a is

1 4 9 16

and b is

4 7 9 9 11

then `mergeSorted` returns the array list

1 4 4 7 9 9 9 11 16

- ■ Business P6.29 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.



Implement a method

```
public static void discount(double[] prices, boolean[] isPet, int nItems)
```

The method receives information about a particular sale. For the *i*th item, `prices[i]` is the price before any discount, and `isPet[i]` is true if the item is a pet.

Write a program that prompts a cashier to enter each price and then a Y for a pet or N for another item. Use a price of -1 as a sentinel. Save the inputs in an array. Call the method that you implemented, and display the discount.

- ■ Business P6.30 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 customer's purchase amount is stored in an `ArrayList<Double>` and the customer's name is stored in a corresponding `ArrayList<String>`.

Implement a method

```
public static String nameOfBestCustomer(ArrayList<Double> sales,
      ArrayList<String> customers)
```

that returns 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 two array lists, calls the method that you implemented, and displays the result. Use a price of 0 as a sentinel.

- ■ ■ Business P6.31 Improve the program of Exercise P6.30 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 static ArrayList<String> nameOfBestCustomers(ArrayList<Double> sales,
      ArrayList<String> customers, int topN)
```

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

- ■ Science P6.32 Sounds can be represented by an array of “sample values” that describe the intensity of the sound at a point in time. The program `ch06/sound/SoundEffect.java` reads a sound file (in WAV format), calls a method `process` for processing the sample values, and saves the sound file. Your task is to implement the `process` method by introducing an echo. For each sound value, add the value from 0.2 seconds ago. Scale the result so that no value is larger than 32767.



- ■ ■ Science P6.33 You are given a two-dimensional array of values that give the height of a terrain at different points in a square. Write a method

```
public static void floodMap(double[][] heights, double waterLevel)
```

that prints out a flood map, showing which of the points in the terrain would be flooded if the water level was the given value. In the flood map, print a `*` for each flooded point and a space for each point that is not flooded.

Here is a sample map:

```

* * * *      * *
* * * * *    * * *
* * * *      * *
* * *      * * *
* * * *      * * * *
* * * * * * * * *
* *      * * *
*          * * * * *
                    * *
                    * * *

```



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

- ■ Science P6.34 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). Implement a method

```
public static void smooth(double[] values, int size)
```

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

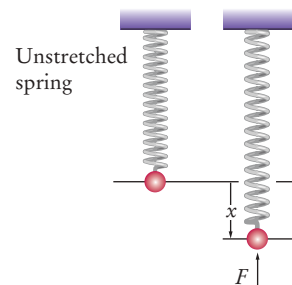
- ■ Science P6.35 Modify the `ch06/animation/BlockAnimation.java` program to show an animated sine wave. In the i th frame, shift the sine wave by i degrees.

- ■ ■ Science P6.36 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

$$F = -kx$$

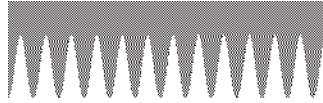
where k is a constant that depends on the spring. (Use 10 N/m for this simulation.)

Start with a given displacement x (say, 0.5 meter). Set the initial velocity v to 0. Compute the acceleration a

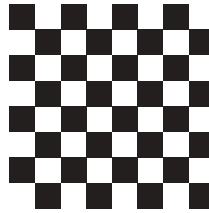


from Newton's law ($F = ma$) and Hooke's law, using a mass of 1 kg. Use a small time interval $\Delta t = 0.01$ second. Update the velocity—it changes by $a\Delta t$. Update the displacement—it changes by $v\Delta t$.

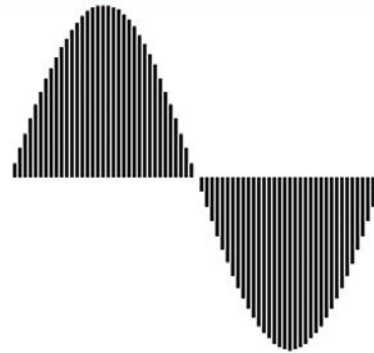
Every ten iterations, plot the spring displacement as a bar, where 1 pixel represents 1 cm. Use the technique in Special Topic 4.3 for creating an image.



- **Graphics P6.37** Using the technique of Special Topic 4.3, generate the image of a checkerboard.



- **Graphics P6.38** Using the technique of Special Topic 4.3, generate the image of a sine wave. Draw a line of pixels for every five degrees.



ANSWERS TO SELF-CHECK QUESTIONS

- `int[] primes = { 2, 3, 5, 7, 11 };`
- 2, 3, 5, 3, 2
- 3, 4, 6, 8, 12
- `values[0] = 10;`
`values[9] = 10;`
or better: `values[values.length - 1] = 10;`
- `String[] words = new String[10];`
- `String[] words = { "Yes", "No" };`
- No. Because you don't store the values, you need to print them when you read them. But you don't know where to add the `<=` until you have seen all values.
- It counts how many elements of `values` are zero.
- ```
for (double x : values)
{
 System.out.println(x);
}
```
- ```
double product = 1;
for (double f : factors)
{
    product = product * f;
}
```

11. The loop writes a value into `values[i]`. The enhanced for loop does not have the index variable `i`.

```
12. 20 <= largest value
    10
    20 <= largest value
```

```
13. int count = 0;
    for (double x : values)
    {
        if (x == 0) { count++; }
    }
```

14. If all elements of `values` are negative, then the result is incorrectly computed as 0.

```
15. for (int i = 0; i < values.length; i++)
    {
        System.out.print(values[i]);
        if (i < values.length - 1)
        {
            System.out.print(" | ");
        }
    }
```

Now you know why we set up the loop the other way.

16. If the array has no elements, then the program terminates with an exception.

17. If there is a match, then `pos` is incremented before the loop exits.

18. This loop sets all elements to `values[pos]`.

```
19. int[] numbers = squares(5);
```

```
20. public static void fill(int[] values, int value)
    {
        for (int i = 0; i < values.length; i++)
        {
            values[i] = value; }
    }
```

21. The method returns an array whose length is given in the first argument. The array is filled with random integers between 0 and `n - 1`.

22. The contents of `scores` is unchanged. The `reverse` method returns a new array with the reversed numbers.

23.

| values | result | i |
|-----------|----------------------|--------------|
| [1, 4, 9] | [0, 0, 0] | 0 |
| | [9, 0, 0] | 1 |
| | [9, 4, 0] | 2 |
| | [9, 4, 1] | |

24. Use the first algorithm. The order of elements does not matter when computing the sum.

25. Find the minimum value.

Calculate the sum.

Subtract the minimum value.

26. Use the algorithm for counting matches (Section 4.7.2) twice, once for counting the positive values and once for counting the negative values.

27. You need to modify the algorithm in Section 6.3.4.

```
boolean first = true;
for (int i = 0; i < values.length; i++)
{
    if (values[i] > 0)
    {
        if (first) { first = false; }
        else { System.out.print(", "); }
    }
    System.out.print(values[i]);
}
```

Note that you can no longer use `i > 0` as the criterion for printing a separator.

28. Use the algorithm to collect all positive elements in an array, then use the algorithm in Section 6.3.4 to print the array of matches.

29. The paperclip for `i` assumes positions 0, 1, 2, 3. When `i` is incremented to 4, the condition `i < size / 2` becomes false, and the loop ends. Similarly, the paperclip for `j` assumes positions 4, 5, 6, 7, which are the valid positions for the second half of the array.



30. It reverses the elements in the array.

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

```
i = 0
j = size - 1
```

```
While (i < j)
```

```
  If (a[i] is odd)
```

```
    Swap elements at positions i and j.
```

```
    j--
```

```
  Else
```

```
    i++
```

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

```
i = 0
```

```
moved = size
```

```
While (i < moved)
```

```
  If (a[i] is odd)
```

```
    Remove the element at position i and add it  
    at the end.
```

```
    moved--
```

- 33.** When you read inputs, you get to see values one at a time, and you can't peek ahead. Picking cards one at a time from a deck of cards simulates this process better than looking at a sequence of items, all of which are revealed.
- 34.** You get the total number of gold, silver, and bronze medals in the competition. In our example, there are four of each.
- 35.**
- ```
for (int i = 0; i < 8; i++)
{
 for (int j = 0; j < 8; j++)
 {
 board[i][j] = (i + j) % 2;
 }
}
```

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

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

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

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

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

```
41. "Ann", "Cal"
```

**42.** The names variable has not been initialized.

**43.** names1 contains "Emily", "Bob", "Cindy", "Dave"; names2 contains "Dave"

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

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

**45.** Reading inputs into an array list is much easier.