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; 1
```

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]; 2
```

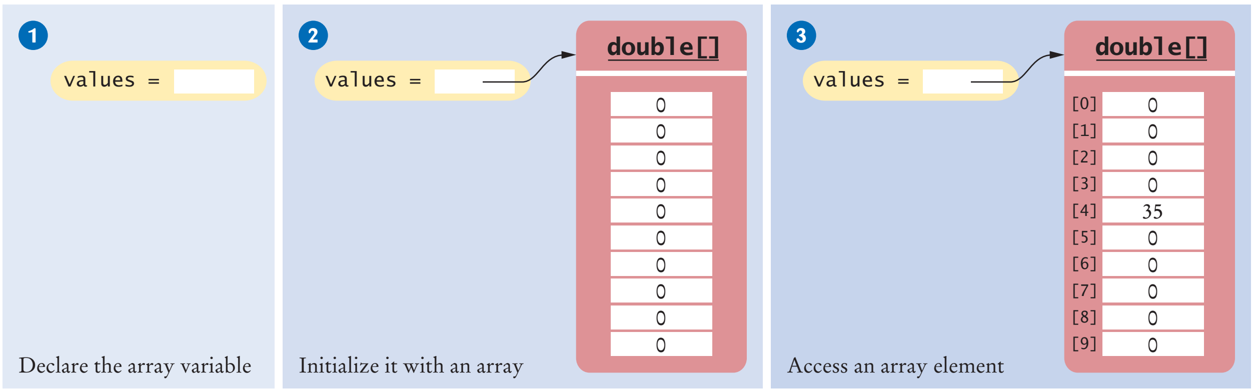


Figure 1 An Array of Size 10

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.

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.

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
```

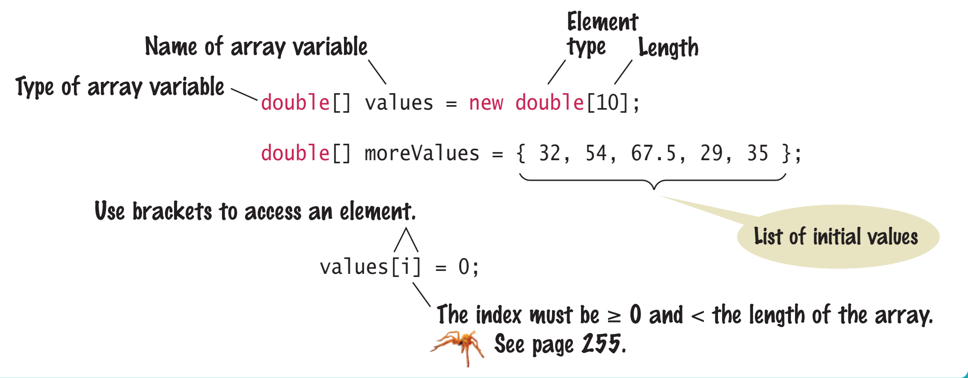
Now the number 4 slot of `values` is filled with 35 (see Figure 1). This "slot number" is called an *index*. Each slot in an array contains an *element*.

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]);
```

Syntax 6.1 Arrays

Syntax	To construct an array:	`new` *typeName*`[`*length*`]`
	To access an element:	*arrayReference*`[`*index*`]`



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

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

values[i] = 0;
```

Name of array variable
Element type
Length
Type of array variable
List of initial values
Use brackets to access an element.
The index must be ≥ 0 and < the length of the array. See page 255.

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

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.

An array index must be at least zero and less than the size of the array.

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.

A bounds error, which occurs if you supply an invalid array index, can cause your program to terminate.

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`.

Table 1 Declaring Arrays

`int[] numbers = new int[10];`	An array of ten integers. All elements are initialized with zero.

`final int LENGTH = 10;`
`int[] numbers = new int[LENGTH];`
	It is a good idea to use a named constant instead of a "magic number".

`int length = in.nextInt();`
`double[] data = new double[length];`
	The length need not be a constant.

`int[] squares = { 0, 1, 4, 9, 16 };`	An array of five integers, with initial values.
`String[] friends = { "Emily", "Bob", "Cindy" };`	An array of three strings.
🚫 `double[] data = new int[10];`	**Error:** You cannot initialize a `double[]` variable with an array of type `int[]`.

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

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

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

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

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

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.

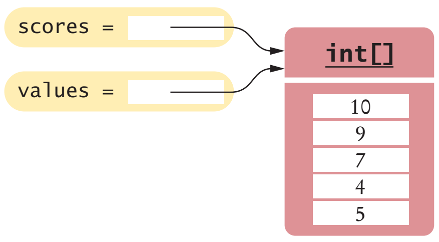


Figure 2
Two Array Variables Referencing the Same Array

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:

ONLINE EXAMPLE

A program demonstrating array operations.

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

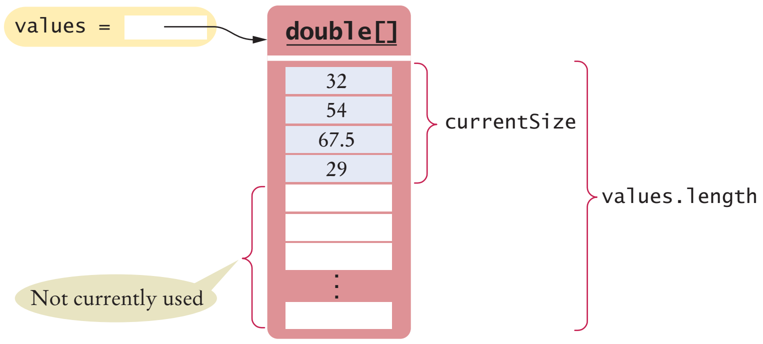


Figure 3 A Partially Filled Array

SELF CHECK

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

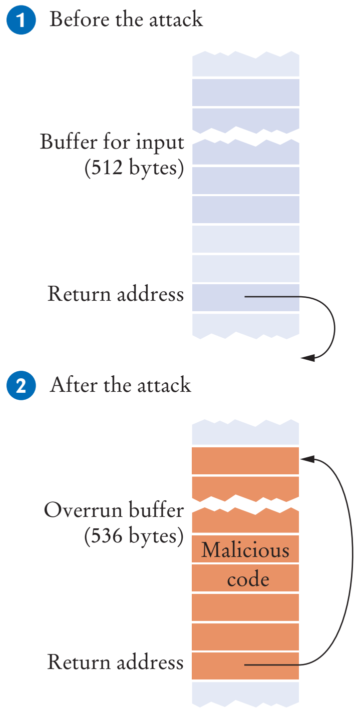
In November 1988, Robert Morris, a student at Cornell University, launched a so-called virus program that infected about 6,000 computers connected to the Internet across the United States. Tens of thousands of computer users were unable to read their e-mail or otherwise use their computers. All major 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.



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.

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

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:

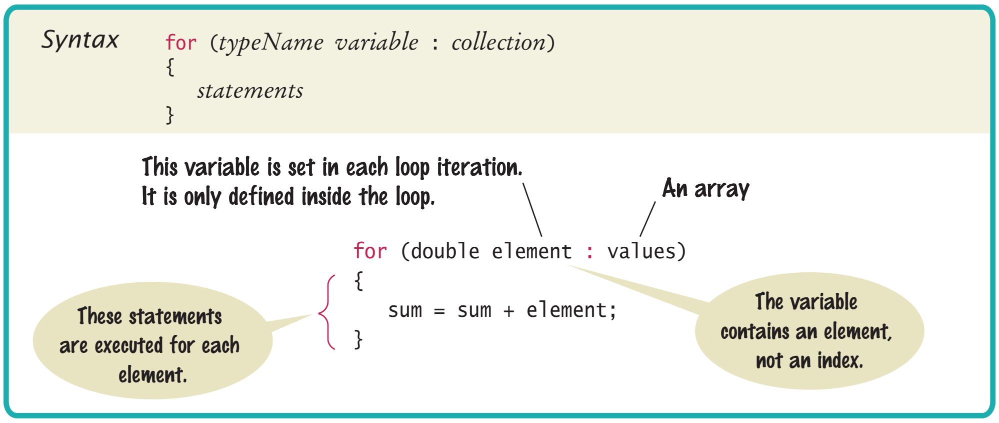
```
for (int i = 0; i < values.length; i++)
{
   values[i] = 0; // OK
}
```

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
}
```

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

This variable is set in each loop iteration. It is only defined inside the loop.
An array
These statements are executed for each element.
The variable contains an element, not an index.

SELF CHECK

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]);
}
```

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]
```

To print five elements, you need four separators.

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 `current-Size` variable. (See Figure 4.)

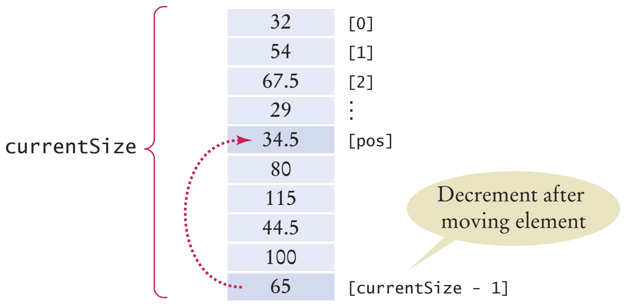


Figure 4
Removing an Element in an Unordered Array

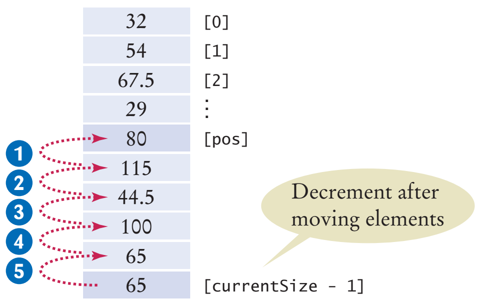


Figure 5
Removing an Element in an Ordered Array

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

ANIMATION
Removing from an Array

The situation is more complex if the order of the elements matters. Then you must move all elements following the element to be removed to a lower index, and then decrement the variable holding the size of the array. (See Figure 5.)

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

6.3.7 Inserting an Element

ANIMATION
Inserting into an Array

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).

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

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;
}
```

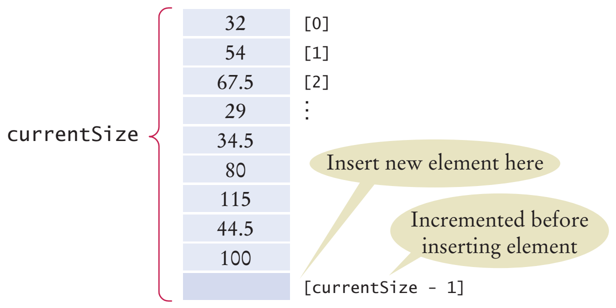


Figure 6
Inserting an Element in an Unordered Array

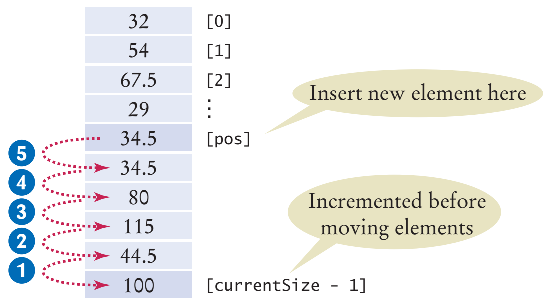


Figure 7
Inserting an Element in an Ordered Array

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:

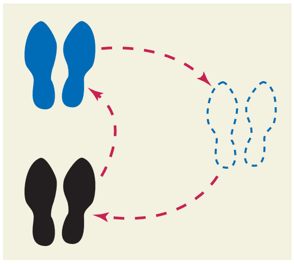
Use a temporary variable when swapping two elements.

```
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.

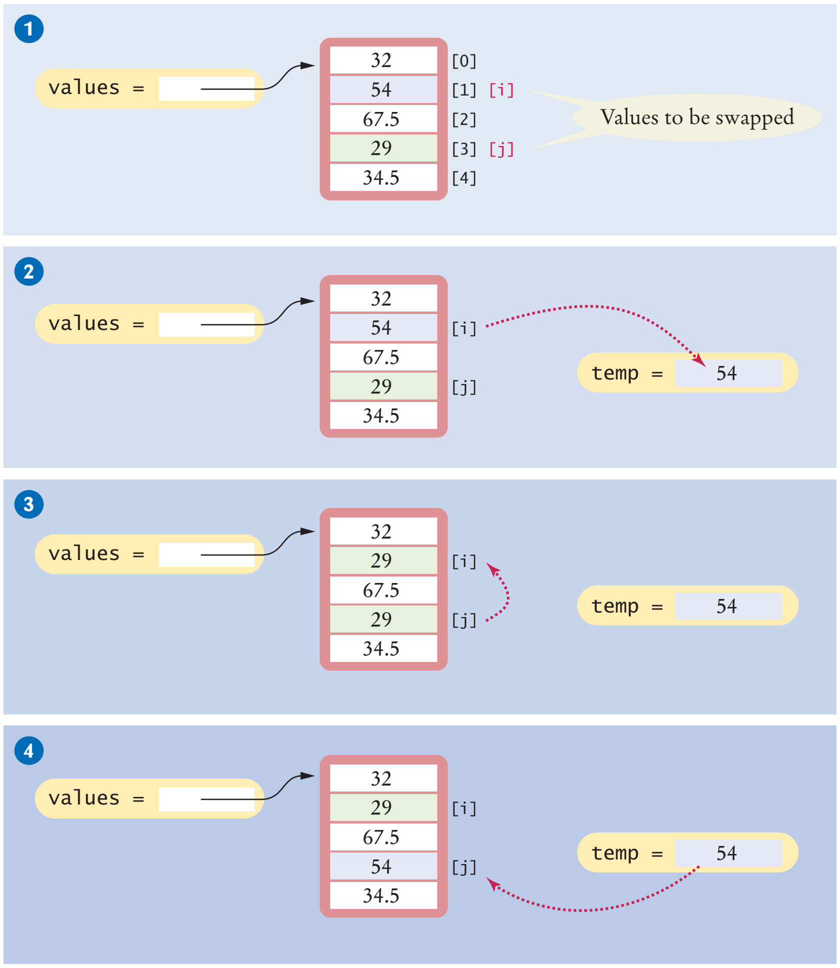


Figure 8 Swapping Array Elements

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; 1
```

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); 2
```

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.

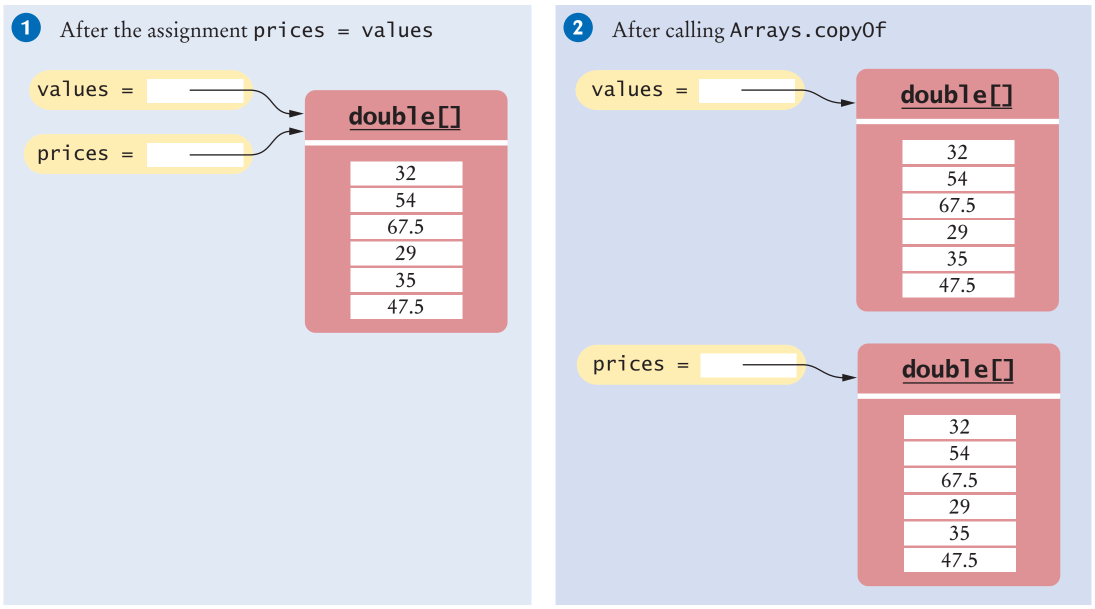


Figure 9 Copying an Array Reference versus Copying an Array

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); 1
values = newValues; 2
```

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

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

with

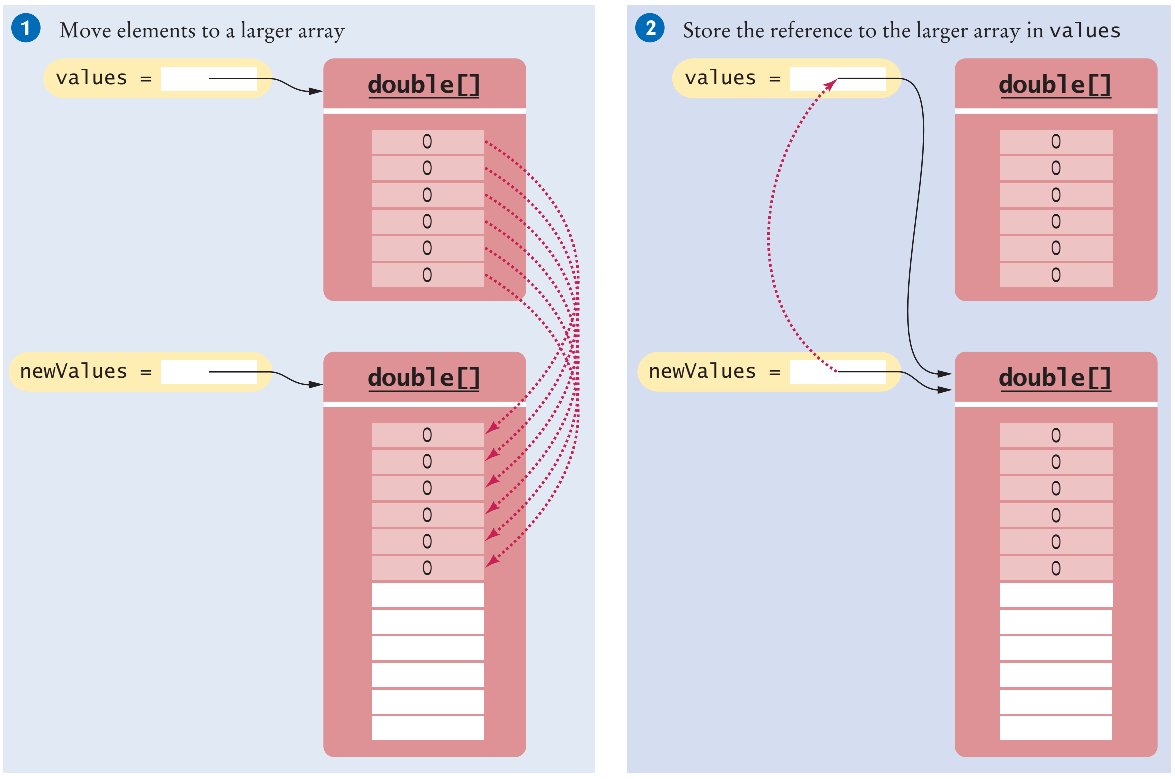

1 Move elements to a larger array
values =
double[]
0
0
0
0
0
0
newValues =
double[]
0
0
0
0
0
0
2 Store the reference to the larger array in values
values =
double[]
newValues =
double[]

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

```
import java.util.Scanner;

/**
   This program reads a sequence of values and prints them, marking the largest value.
*/
public class LargestInArray
{
   public static void main(String[] args)
   {
      final int LENGTH = 100;
      double[] values = new double[LENGTH];
      int currentSize = 0;

      // Read inputs

      System.out.println("Please enter values, Q to quit:");
      Scanner in = new Scanner(System.in);
      while (in.hasNextDouble() && currentSize < values.length)
      {
         values[currentSize] = in.nextDouble();
         currentSize++;
      }

      // Find the largest value

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

      // Print all values, marking the largest

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

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.

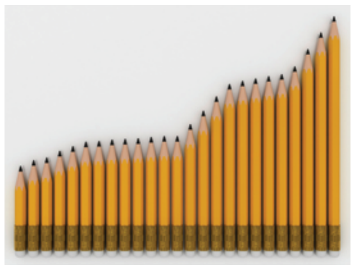
Common Error 6.3

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.

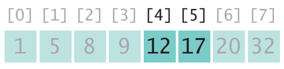
Consider the following sorted array values.

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

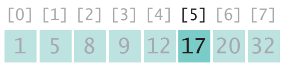
We would like to see whether the number 15 is in the array. Let's narrow our search by finding whether the number is in the first or second half of the array. The last point in the first half of the values array, values[3], is 9, which is smaller than the number we are looking for. Hence, we should look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



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:



We still don't have a match because 15 ≠ 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

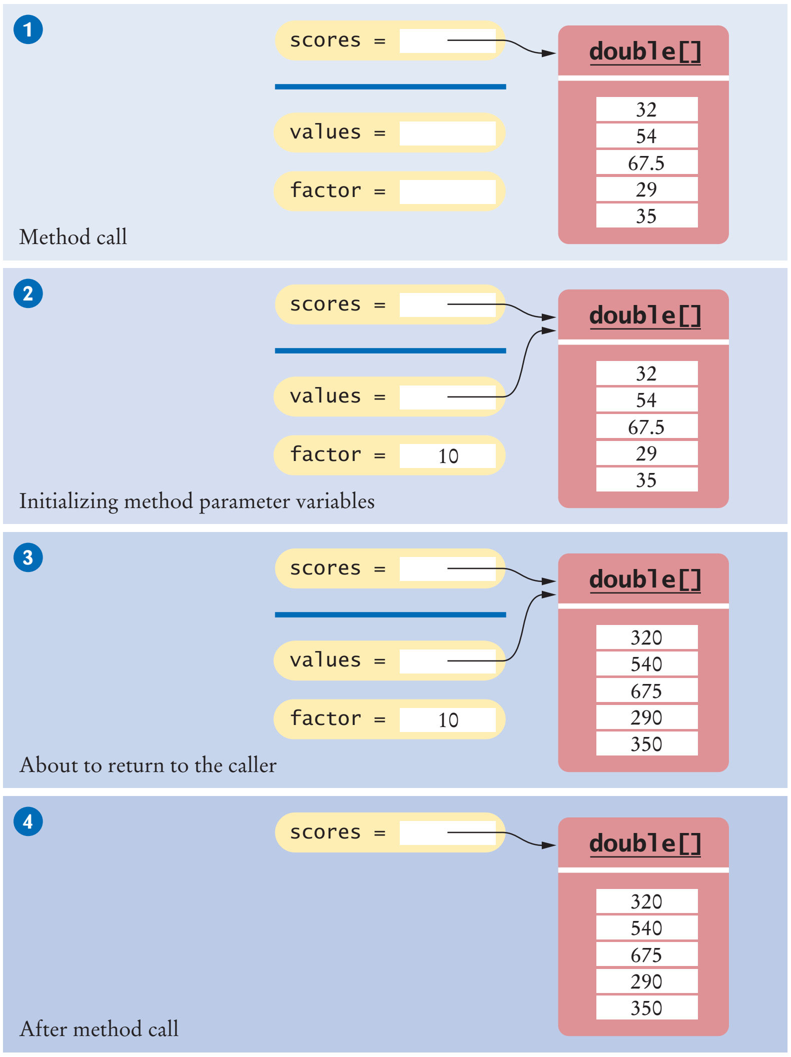


Figure 11 Trace of Call to the `multiply` Method

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

```
import java.util.Scanner;

/**
   This program reads, scales, and reverses a sequence of numbers.
*/
public class Reverse
{
   public static void main(String[] args)
   {
      double[] numbers = readInputs(5);
      multiply(numbers, 10);
      printReversed(numbers);
   }

   /**
      Reads a sequence of floating-point numbers.
      @param numberOfInputs the number of inputs to read
      @return an array containing the input values
   */
   public static double[] readInputs(int numberOfInputs)
   {
     System.out.println("Enter " + numberOfInputs + " numbers: ");
     Scanner in = new Scanner(System.in);
     double[] inputs = new double[numberOfInputs];
     for (int i = 0; i < inputs.length; i++)
     {
        inputs[i] = in.nextDouble();
     }
     return inputs;
   }

   /**
      Multiplies all elements of an array by a factor.
      @param values an array
      @param factor the value with which element is multiplied
   */
   public static void multiply(double[] values, double factor)
   {
      for (int i = 0; i < values.length; i++)
      {
         values[i] = values[i] * factor;
      }
   }

   /**
      Prints an array in reverse order.
      @param values an array of numbers
      @return an array that contains the elements of values in reverse order
   */
   public static void printReversed(double[] values)
   {
      // Traverse the array in reverse order, starting with the last element
      for (int i = values.length - 1; i >= 0; i--)
      {
         System.out.print(values[i] + " ");
      }
      System.out.println();
   }
}
```

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.

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

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;
   }
```

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”.

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