Note that, in only two comparisons, we have reduced the viable candidates from 15 items down to 3 items. Employing the same approach again, we select the middle element, 67, and find the element we are seeking. If it had not been our target, we would have continued with this process until we either found the value or eliminated all possible data.

With each comparison, a binary search eliminates approximately half of the remaining data to be searched (it also eliminates the middle element as well). Therefore, a binary search eliminates half of the data with the first comparison, another quarter of the data with the second comparison, another eighth of the data with the third comparison, and so on.

KEY CONCEPT

A binary search eliminates half of the viable candidates with each comparison.

The static `binarySearch` method of the `Sorting` class in Listing 13.3 implements a binary search. Like the `linearSearch` method, it accepts an array of `Comparable` objects to be searched and the target value being sought.

The values of the integer variables `first` and `last` represent the range of indexes that make up the viable candidates at any given point during the algorithm. Initially, they are set to encompass the entire array.

Similar to the linear search, the `while` loop in the `binarySearch` method continues until the target element is found or the viable data is exhausted. If the target is found, the value of `result` is set and the loop terminates. On the other hand, if the value of `first` ever exceeds the value of `last`, then all viable candidates have been eliminated, the loop terminates, and a `null` value is returned.

In each iteration of the loop, the midpoint of the viable candidates is calculated by taking the average of the `first` and `last` indexes (using integer division). If the target value is found, the return value `result` is set to the original element from the array. Otherwise, the value of `first` or `last` is adjusted to either side of the midpoint, depending on which half of the candidates remains viable, and the search continues.

This version of the binary search algorithm is implemented iteratively. It can also be implemented recursively, which is left as a programming project.

In general, a binary search is more efficient than a linear search because it eliminates many candidates with each comparison. But a binary search requires that the list be in sorted order, whereas a linear search does not. A formal comparison of the efficiency of these search algorithms is presented in the last section of this chapter.

13.2 Sorting

Sorting is the process of arranging a group of items into a defined order, either ascending or descending, based on some criteria. For example, you may want to alphabetize a list of names or put a list of survey results into descending numeric

KEY CONCEPT
Sorting is the process of arranging a list of items into a defined order based on some criteria.

order. Many sort algorithms have been developed and critiqued over the years. In fact, sorting is considered to be a classic area of study in computer science.

In this section we examine five different sorting algorithms: selection sort, insertion sort, bubble sort, quick sort, and merge sort. The first three are roughly equal in terms of efficiency, but approach the problem from different perspectives. The last two are much more efficient, though a bit more complicated. The last section of this chapter explores ways to examine the efficiency of algorithms, and formally compares these sorting algorithms.

As we did with our search algorithms, we currently restrict ourselves to sorting an array of objects. Other sorting techniques are examined later in the book based on alternative ways to structure our data.

The program in Listing 13.4 creates an array of `Contact` objects just like the array used in our example of search algorithms in Listing 13.2. This time, however, after populating the array, we invoke a method that sorts the elements and then prints them out in order. This example uses the same `Contact` class shown back in Listing 13.1 and relies on its implementation of the `Comparable` interface to determine the relative order of the objects. Note that when two `Contact` objects contain the same last name, the first name is used to determine the order.

The `SortPlayerList` program invokes the static `selectionSort` method of the `Sorting` class, shown in Listing 13.5 on page 492. However, any of the sorting methods in that class could be used to sort the list of `Contact` objects by changing the name of the method invoked.

Let's explore each of these algorithms in turn.

Selection Sort

KEY CONCEPT
The selection sort algorithm sorts a list of values by repeatedly putting a particular value into its final, sorted position.

The *selection sort* algorithm sorts a list of values by repetitively putting a particular value into its final, sorted position. In other words, for each position in the list, the algorithm selects the value that should go in that position and then puts it there.

The general strategy of the selection sort algorithm is as follows: Scan the entire list to find the smallest value. Exchange that value with the value in the first position of the list. Scan the rest of the list (all but the first value) to find the smallest value, and then exchange it with the value in the second position of the list. Scan the rest of the list (all but the first two values) to find the smallest value, and then exchange it with the value in the third position

LISTING 13.4

```
//********************************************************************
//  SortPlayerList.java       Java Foundations
//
//  Demonstrates a selection sort of Comparable objects.
//********************************************************************

public class SortPlayerList
{
   //-----------------------------------------------------------------
   //  Creates an array of Contact objects, sorts them, then prints
   //  them.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      Contact[] players = new Contact[7];

      players[0] = new Contact ("Rodger", "Federer", "610-555-7384");
      players[1] = new Contact ("Andy", "Roddick", "215-555-3827");
      players[2] = new Contact ("Maria", "Sharapova", "733-555-2969");
      players[3] = new Contact ("Venus", "Williams", "663-555-3984");
      players[4] = new Contact ("Lleyton", "Hewitt", "464-555-3489");
      players[5] = new Contact ("Eleni", "Daniilidou", "322-555-2284");
      players[6] = new Contact ("Serena", "Williams", "243-555-2837");

      Sorting.selectionSort(players);

      for (Comparable player: players)
         System.out.println (player);
   }
}
```

OUTPUT

```
Daniilidou, Eleni: 322-555-2284
Federer, Rodger: 610-555-7384
Hewitt, Lleyton: 464-555-3489
Roddick, Andy: 215-555-3827
Sharapova, Maria: 733-555-2969
Williams, Serena: 243-555-2837
Williams, Venus: 663-555-3984
```

LISTING 13.5

```
//********************************************************************
//  Sorting.java       Java Foundations
//
//  Contains various sort algorithms that operate on an array of
//  Comparable objects.
//********************************************************************

public class Sorting
{
   //-----------------------------------------------------------------
   //  Sorts the specified array of integers using the selection
   //  sort algorithm.
   //-----------------------------------------------------------------
   public static void selectionSort (Comparable[] data)
   {
      int min;

      for (int index = 0; index < data.length-1; index++)
      {
         min = index;
         for (int scan = index+1; scan < data.length; scan++)
            if (data[scan].compareTo(data[min]) < 0)
               min = scan;

         swap (data, min, index);
      }
   }

   //-----------------------------------------------------------------
   //  Swaps two elements in the specified array.
   //-----------------------------------------------------------------
   private static void swap (Comparable[] data, int index1, int index2)
   {
       Comparable temp = data[index1];
       data[index1] = data[index2];
       data[index2] = temp;
   }

   //-----------------------------------------------------------------
   //  Sorts the specified array of objects using an insertion
   //  sort algorithm.
   //-----------------------------------------------------------------
```

LISTING 13.5 *continued*

```
public static void insertionSort (Comparable[] data)
{
   for (int index = 1; index < data.length; index++)
   {
      Comparable key = data[index];
      int position = index;

      //  Shift larger values to the right
      while (position > 0 && data[position-1].compareTo(key) > 0)
      {
         data[position] = data[position-1];
         position--;
      }

      data[position] = key;
   }
}

//-----------------------------------------------------------------
//  Sorts the specified array of objects using a bubble sort
//  algorithm.
//-----------------------------------------------------------------
public static void bubbleSort (Comparable[] data)
{
   int position, scan;

   for (position = data.length - 1; position >= 0; position--)
   {
      for (scan = 0; scan <= position - 1; scan++)
         if (data[scan].compareTo(data[scan+1]) > 0)
            swap (data, scan, scan+1);
   }
}

//-----------------------------------------------------------------
//  Sorts the specified array of objects using the quick sort
//  algorithm.
//-----------------------------------------------------------------
public static void quickSort (Comparable[] data, int min, int max)
{
   int pivot;

   if (min < max)
```

LISTING 13.5 *continued*

```
      {
         pivot = partition (data, min, max);  // make partitions
         quickSort(data, min, pivot-1);  // sort left partition
         quickSort(data, pivot+1, max);  // sort right partition
      }
   }

   //-----------------------------------------------------------------
   //  Creates the partitions needed for quick sort.
   //-----------------------------------------------------------------
   private static int partition (Comparable[] data, int min, int max)
   {
      //  Use first element as the partition value
      Comparable partitionValue = data[min];

      int left = min;
      int right = max;

      while (left < right)
      {
         // Search for an element that is > the partition element
         while (data[left].compareTo(partitionValue) <= 0 && left < right)
            left++;

         //  Search for an element that is < the partitionelement
         while (data[right].compareTo(partitionValue) > 0)
            right--;

         if (left < right)
            swap(data, left, right);
      }

      //  Move the partition element to its final position
      swap (data, min, right);

      return right;
   }

   //-----------------------------------------------------------------
   //  Sorts the specified array of objects using the merge sort
   //  algorithm.
   //-----------------------------------------------------------------
```

LISTING 13.5 *continued*

```
public static void mergeSort (Comparable[] data, int min, int max)
{
   if (min < max)
   {
      int mid = (min + max) / 2;
      mergeSort (data, min, mid);
      mergeSort (data, mid+1, max);
      merge (data, min, mid, max);
   }
}

//-----------------------------------------------------------------
//  Sorts the specified array of objects using the merge sort
//  algorithm.
//-----------------------------------------------------------------
public static void merge (Comparable[] data, int first, int mid,
   int last)
{
   Comparable[] temp = new Comparable[data.length];

   int first1 = first, last1 = mid;  // endpoints of first subarray
   int first2 = mid+1, last2 = last;  // endpoints of second subarray
   int index = first1;  // next index open in temp array

   //  Copy smaller item from each subarray into temp until one
   //  of the subarrays is exhausted
   while (first1 <= last1 && first2 <= last2)
   {
     if (data[first1].compareTo(data[first2]) < 0)
     {
        temp[index] = data[first1];
        first1++;
     }
     else
     {
        temp[index] = data[first2];
        first2++;
     }
     index++;
   }
```

LISTING 13.5 *continued*

```
      //  Copy remaining elements from first subarray, if any
      while (first1 <= last1)
      {
         temp[index] = data[first1];
         first1++;
         index++;
      }

      //  Copy remaining elements from second subarray, if any
      while (first2 <= last2)
      {
         temp[index] = data[first2];
         first2++;
         index++;
      }

      //  Copy merged data into original array
      for (index = first; index <= last; index++)
         data[index] = temp[index];
   }
}
```

of the list. Continue this process for each position in the list. When complete, the list is sorted. The selection sort process is illustrated in Figure 13.3.

The `selectionSort` method accepts an array of `Comparable` objects as a parameter. When control returns to the calling method, the elements within the array are sorted. The method uses two loops to sort an array. The outer loop controls the position in the array where the next smallest value will be stored. The inner loop finds the smallest value in the rest of the list by scanning all positions greater than or equal to the index specified by the outer loop. When the smallest value is determined, it is exchanged with the value stored at `index`.

The exchange of two elements in the array, called *swapping*, is accomplished with a call to a support method called `swap`. Many of the sort algorithms we examine in this chapter swap elements and make use of this method. The `swap` method uses three assignment statements to make the exchange.

Note that because this algorithm finds the smallest value during each iteration, the result is an array sorted in ascending order (i.e., smallest to largest). The algorithm

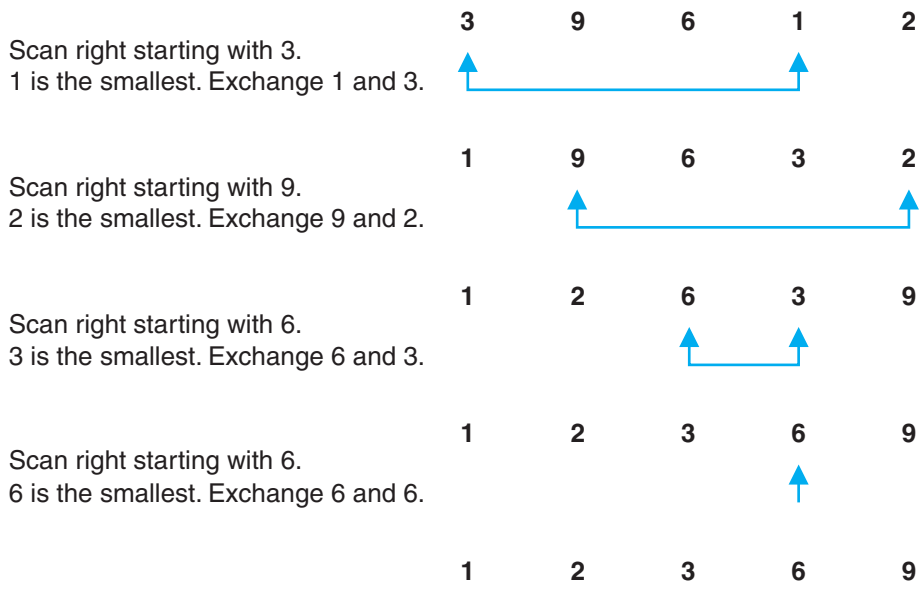

FIGURE 13.3 An example of section sort processing

can easily be changed to put values in descending order by finding the largest value each time through the loop.

Insertion Sort

The *insertion sort* algorithm sorts a list of values by repetitively inserting a particular value into a subset of the list that has already been sorted. One at a time, each unsorted element is inserted at the appropriate position in that sorted subset until the entire list is in order.

KEY CONCEPT
The insertion sort algorithm sorts a list of values by repetitively inserting a particular value into a subset of the list that has already been sorted.

The general strategy of the insertion sort algorithm is as follows: Sort the first two values in the list relative to each other by exchanging them if necessary. Insert the list's third value into the appropriate position relative to the first two (sorted) values. Then, insert the fourth value into its proper position relative to the first three values in the list. Each time an insertion is made, the number of values in the sorted subset increases by one. Continue this process until all values in the list are completely sorted. The insertion process requires that the other values in the array shift to make room for the inserted element. Figure 13.4 on the next page illustrates the insertion sort process.

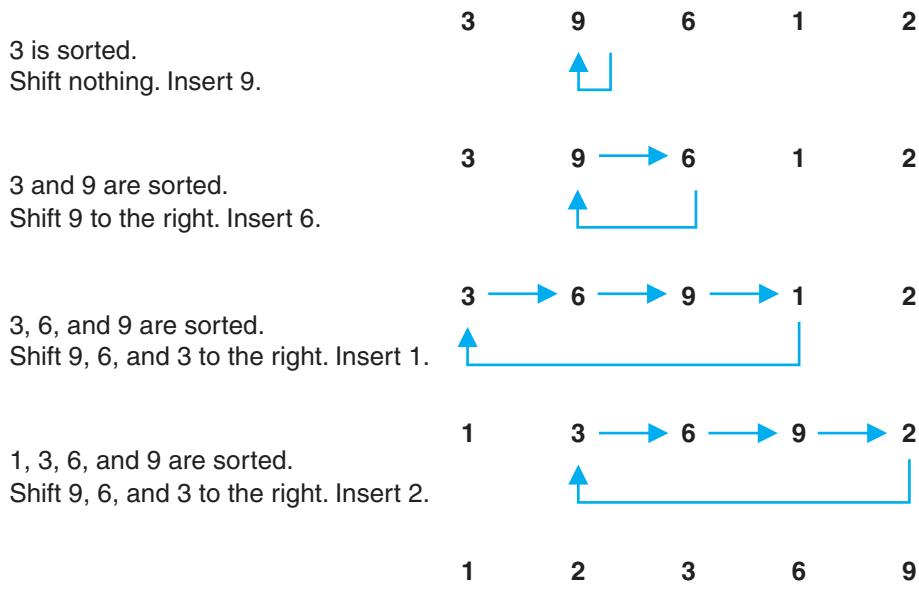

FIGURE 13.4 An example of insertion sort processing

Similar to the selection sort implementation, the `insertionSort` method uses two loops to sort an array of objects. In the insertion sort, however, the outer loop controls the index in the array of the next value to be inserted. The inner loop compares the current insert value with values stored at lower indexes (which make up a sorted subset of the entire list). If the current insert value is less than the value at `position`, then that value is shifted to the right. Shifting continues until the proper position is "opened" to accept the insert value. Each iteration of the outer loop adds one more value to the sorted subset of the list, until the entire list is sorted.

Bubble Sort

A *bubble sort* sorts values by repeatedly comparing neighboring elements in the list and swapping their position if they are not in order relative to each other.

KEY CONCEPT
The bubble sort algorithm sorts a list by repeatedly comparing neighboring elements and swapping them if necessary.

The general strategy of the bubble sort algorithm is as follows: Scan through the list comparing adjacent elements and swap them if they are not in relative order. This has the effect of "bubbling" the largest value to the last position in the list, which is its appropriate position in the final, sorted list. Then scan through the list again, bubbling up the second-to-last value. This process continues until all elements have been bubbled into their correct positions.

Each pass through the bubble sort algorithm moves one value to its final position. A pass may also reposition other elements as well. Let's follow this process for a small list of integers. Suppose we started with this list:

```
9 6 8 12 3 1 7
```

We would first compare 9 and 6 and, finding them not in the correct order, swap them, yielding:

```
6 9 8 12 3 1 7
```

Then, we would compare 9 to 8 and, again, finding them not in the correct order, swap them, yielding:

```
6 8 9 12 3 1 7
```

Then, we would compare 9 to 12. Since they are in the correct order, we don't swap them. Instead, we move on to the next pair of values. That is, we then compare 12 and 3. Since they are not in order, we swap them, yielding:

```
6 8 9 3 12 1 7
```

We then compare 12 to 1 and swap them, yielding:

```
6 8 9 3 1 12 7
```

We then compare 12 to 7 and swap them, yielding:

```
6 8 9 3 1 7 12
```

This completes one pass through the data to be sorted. After this first pass, the largest value in the list (12) is in its correct position. The next pass will bubble the value 9 up to its final position. Each subsequent pass through the data guarantees that one more element is put into the correct position. Thus, we make n–1 passes through the data, because if n–1 elements are in the correct, sorted positions, the *n*th item must also be in the correct location.

The `bubbleSort` method in the `Sorting` class implements this approach. The outer `for` loop represents the n–1 passes through the data. The inner `for` loop scans through the data, performing the pair-wise comparisons of the neighboring data and swapping them if necessary. The support method `swap` is used to perform the exchange of two elements, just as we used it in the `selectionSort` method.

Note that the outer loop also has the effect of decreasing the position that represents the maximum index to examine in the inner loop. That is, after the first pass, which puts the last value in its correct position, there is no need to consider

that value in future passes through the data. After the second pass, we can forget about the last two, and so on. Thus the inner loop examines one less value on each pass.

Quick Sort

The sort algorithms we have discussed thus far in this chapter (selection sort, insertion sort, and bubble sort) are relatively simple, but they are generally inefficient. They each use a pair of nested loops and require roughly n^2 comparisons to sort a list of n elements. Let's now turn our attention to more efficient sorts.

KEY CONCEPT
The quick sort algorithm sorts a list by partitioning the list and then recursively sorting the two partitions.

The *quick sort* algorithm sorts a list by partitioning the list based on an arbitrarily chosen *partition element* and then recursively sorting the sublists on either side of the partition element.

The general strategy of the quick sort algorithm is as follows: First, choose one element of the list to act as a partition element. Next, partition the list so that all elements less than the partition element are to the left of that element and all elements greater than the partition element are to the right. Finally, apply this quick sort strategy (recursively) to both partitions.

If the items to be sorted are in random order initially, the choice of the partition element is arbitrary, and we will use the first element in the list. For efficiency reasons, it would be nice if the partition element divided the list roughly in half, but the algorithm will work no matter what element is chosen as the partition.

Let's look at an example of creating a partition. Suppose we start with the following list:

```
90 65 7 305 120 110 8
```

We choose 90 as our partition element, and then rearrange the list, putting the elements that are less than 90 to the left side and those that are greater than 90 to the right side, yielding two partitions (in bold):

8 65 7 90 **120 110 305**

The relative order of the elements in each partition doesn't matter at this point, as long as every value in the left partition is less than 90 and every value in the right partition is greater than 90. Note that, after creating the two partitions, the partition element (90) is in its final position in the array, and we no longer have to consider or move it again.

We then apply the quick sort algorithm separately to both partitions. This process continues until a partition contains only one element, which is inherently sorted. Thus, after the algorithm is applied recursively to either side, the entire list is sorted.