8.6.2 Selection Sort

The member IDs in the preceding *winners* array were in random order, so when a member was not a winner, our *findWinners* method needed to look at every element in the array before discovering that the ID we were looking for was not in the array. This is not efficient, since most members are not winners. The larger the array, the more inefficient a sequential search becomes. We could simplify the search by arranging the elements in numeric order, which is called **sorting the array.** Once the array is sorted, we can use various algorithms to speed up a search. Later in this chapter, we discuss how to search a sorted array.

In this chapter, we present two basic sorting algorithms, **Selection Sort** and **Insertion Sort.**

Selection Sort derives its name from the algorithm used to sort the array. We select a largest element in the array and place it at the end of the array. Then we select a next-largest element and put it in the next-to-last position in the array. To do this, we consider the unsorted portion of the array as a **subarray.** We repeatedly select a largest value in the current subarray and move it to the end of the subarray, then consider a new subarray by eliminating the elements that are in their sorted locations, until the subarray has only one element. At that time, the array is sorted.

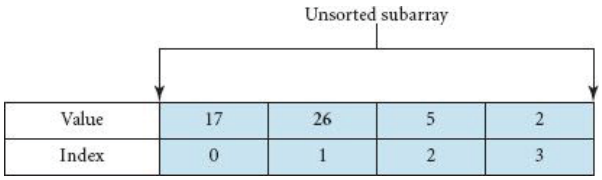
In more formal terms, we can state the Selection Sort algorithm, presented here in pseudocode, in this way:

To sort an array with *n* elements in ascending order:

1. *Consider m elements as a subarray with m = n elements.*
2. *Find the index of a largest value in this subarray.*
3. *Swap the values of the element with the largest value and the element in the last position in the subarray.*
4. *Consider a new subarray of m = m - 1 elements by eliminating the last element in the previous subarray.*
5. *Repeat steps 2 through 4 until m = 1.*

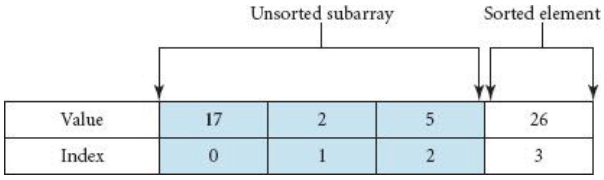
For example, let's walk through a Selection Sort on the following array. At the beginning, the entire array is the subarray (shown here with shading).

We begin by considering the entire array as an unsorted subarray. We find that the largest element is 26 at index 1.



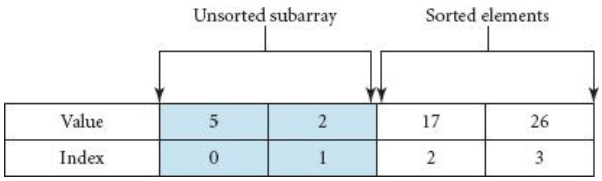
Next we move element 1 to the last element by swapping the values of the elements at indexes 1 and 3.

The value 26 is now in the right place, and we consider elements 0 through 2 as the unsorted subarray.



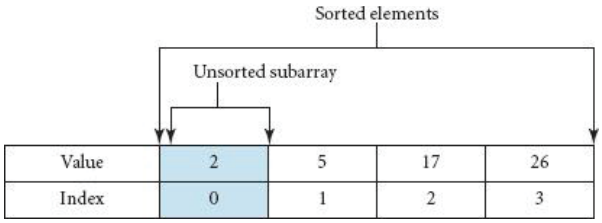
The largest element in the new subarray is 17 at index 0. So we move element 0 to the last index of the subarray (index 2) by swapping the elements at indexes 0 and 2.

The value 17 is now in the right place, and we consider elements 0 and 1 as the new unsorted subarray.



The largest element in the new subarray is 5 at index 0. We move element 0 to the last index of the subarray (index 1) by swapping the elements at indexes 0 and 1.

The value 5 is now in the right place, and we consider element 0 as the new subarray. But because there is only one element in the subarray, the subarray is sorted. Thus the whole array is sorted, and our job is done.



A critical operation in a Selection Sort is swapping two array elements. Before going further, let's examine the algorithm for swapping two array elements.

To swap two values, we need to define a temporary variable that is of the same data type as the values being swapped. This variable will temporarily hold the value of one of the elements, so that we don't lose the value during the swap.

The algorithm, presented here in pseudocode, involves three steps:

To swap elements a and b:

1. *Assign the value of element a to the temporary variable*
2. *Assign the value of element b to element a*
3. *Assign the value in the temporary variable to element b*

For instance, if an array named *array* has *int* elements, and we want to swap the element at index 3 with the element at index 6, we will use the following code:

```
int temp = array[3];      // line 1
array[3] = array[6];      // line 2
array[6] = temp;          // line 3
```

The order of these operations is critical; changing the order might result in loss of data and erroneous data stored in the array.

The following illustrates line by line what happens during the swap:

Before line 1 is executed, our array looks like this:

Value	23	45	7	33	78	90	82	80	90	66
Index	0	1	2	3	4	5	6	7	8	9

Line 1 assigns the value of element 3 to *temp.* After line 1 is executed, the value of *temp* is 33. The array is unchanged.

Value	23	45	7	33	78	90	82	80	90	66
Index	0	1	2	3	4	5	6	7	8	9

33
temp

Line 2 assigns the value of element 6 (82) to element 3. After line 2 is executed, both element 6 and element 3 have the same value. But that's OK, because we saved the value of element 3 in *temp.*

Value	23	45	7	82	78	90	82	80	90	66
Index	0	1	2	3	4	5	6	7	8	9

33
temp

Line 3 assigns the value we saved in *temp* to element 6. After line 3 is executed, the values of elements 3 and 6 have been successfully swapped.

Value	23	45	7	82	78	90	33	80	90	66
Index	0	1	2	3	4	5	6	7	8	9

33
temp

COMMON ERROR TRAP

When swapping elements, be sure to save a value before replacing it with another value to avoid losing data.

Example 8.16 shows the *Sorter* class, which provides a *static selectionSort* method for an integer array.

```
1 /* Sort Utility Class
2 *  Anderson, Franceschi
```

```
 3 */
 4
 5 public class Sorter
 6 {
 7   /**  Uses Selection Sort to sort
 8   *      an integer array in ascending order
 9   *    @param array the array to sort
10   */
11   public static void selectionSort( int [ ] array )
12   {
13     int temp; // temporary location for swap
14     int max;  // index of maximum value in subarray
15
16     for ( int i = 0; i < array.length; i++ )
17     {

18     // find index of largest value in subarray
19     max = indexOfLargestElement( array, array.length - i );
20
21     // swap array[max] and array[array.length - i - 1]
22     temp = array[max];
23     array[max] = array[array.length - i - 1];
24     array[array.length - i - 1] = temp;
25   }
26  }
27
28  /**  Finds index of largest element
29  *    @param    size  the size of the subarray
30  *    @param    array the array to search
31  *    @return   the index of the largest element in the subarray
32  */
33  privatestatic int indexOfLargestElement( int [ ] array, int size )
34  {
35    int index = 0;
36    for( int i = 1; i < size; i++ )
37    {
38       if ( array[i] > array[index] )
39          index = i;
40    }
41    return index;
42   }
43 }
```

EXAMPLE 8.16 The *Sorter* Class

Part of the Selection Sort algorithm is finding the index of the largest element in a subarray, so we implement the Selection Sort with two methods. At lines 7-26, is the *selectionSort* method, which implements the Selection Sort algorithm. To perform its work, the *selectionSort* method calls the utility method, *indexOfLargestElement* (lines 28-42), which returns the index of the largest element in a subarray. This method uses the algorithm discussed earlier in the chapter for finding a maximum value in an array. We declare this method *private* because its only function is to provide a service to the *selectionSort* method. The *indexOfLargestElement* method must also be declared as *static* because the *selectionSort* method is *static,* and thus can call only *static* methods.

In Example 8.17, the client code instantiates an integer array and prints the array before and after the Selection Sort is performed. Because *selectionSort* is a *static* method, we call it using the *Sorter* class name. The output of a sample run is shown in Figure 8.22.

```
 1 /** Client for Selection Sort
 2 *   Anderson, Franceschi
 3 */
 4 import java.util.Random;
 5
 6 public class SelectionSortClient
 7 {
 8   public static void main( String [ ] args )
 9   {
10     // instantiate an array and fill with random values
11     int [ ] numbers = new int [6];
12     Random rand = newRandom( );
13     for ( int i = 0; i < numbers.length; i++ )
14     {
15       numbers[i] = rand.nextInt( 5000 ) + 1;
16     }
17
18     System.out.println( "Before Selection Sort, the array is" );
19     for ( int i = 0; i < numbers.length; i++ )
20        System.out.print( numbers[i] + "\t" );
21     System.out.println( );
22
23     Sorter.selectionSort( numbers ); // sort the array
24
25     System.out.println( "\nAfter Selection Sort, the array is" );
26     for ( int i = 0; i < numbers.length; i++ )
27        System.out.print( numbers[i] + "\t" );
```

```
28      System.out.println( );
29    }
30 }
```

EXAMPLE 8.17 Using Selection Sort

Figure 8.22 Using Selection Sort

```
Before Selection Sort, the array is
3394    279     1181    2471    3660    221

After Selection Sort, the array is
221     279     1181    2471    3394    3660
```

8.6.3 Insertion Sort

Like Selection Sort, Insertion Sort also derives its name from the algorithm used to sort the array. The basic approach to an Insertion Sort is to sort elements much like a card player arranges the cards in sorted order in his or her hand. The player inserts cards one at a time in such a way that the cards on the left side of his or her hand are sorted at all times; the cards on the right side of his or her hand have not yet been inserted into the sorted part of the hand. As Figure 8.23a shows, the three yellow cards on the left (3, 5, and 9) are already arranged in sorted order, and the white cards on the right (4, 2, and 8) have yet to be inserted into their correct location. Note that the "sorted" yellow cards on the left side are not necessarily in their final position yet. We will now insert the 4. We first compare it to the 9; since 4 is smaller than 9, we shift the 9 to the right (Figure 8.23b). We then compare the 4 to the 5; since 4 is smaller than 5, we shift the 5 to the right (Figure 8.23c). We then compare the 4 to the 3; since 4 is larger than 3, the 3 stays in place and we insert the 4 in the empty slot (Figure 8.23d). We are now ready to insert the next card, the 2.

Figure 8.23a The next card to insert is a 4

Figure 8.23b 9 is shifted to the right

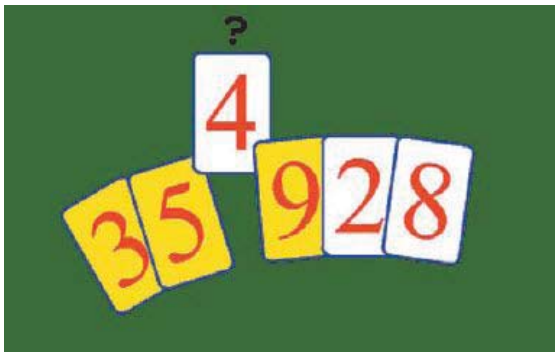


Figure 8.23c 5 is shifted to the right

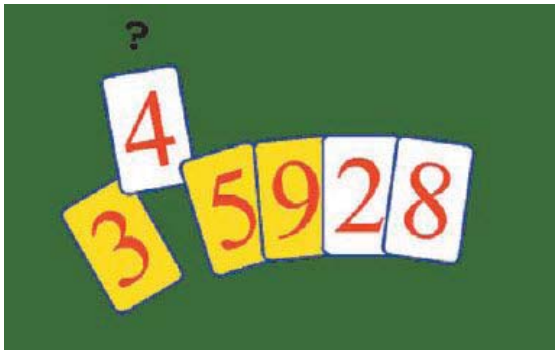


Figure 8.23d 4 is inserted

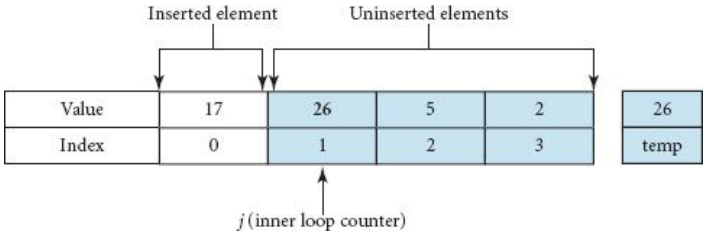
To sort an array of *n* elements in ascending order, Insertion Sort implements a double loop:

- The outer loop executes *n* - 1 times and iterates through the array elements from indexes 1 through *n* - 1. If the variable *i* represents the counter of the outer loop, the array can be thought of as made of three parts:
- a sorted subarray (although the elements may not be in their final position yet) from index 0 to *i* - 1,
- the array element (at index *i*) that we are currently inserting,
- a subarray (from index *i* + 1 to *n* - 1) of elements that have not yet been inserted.
- At each iteration of the outer loop, we insert the current array element at its proper place within the sorted subarray. The inner loop compares the current array element to the elements of the sorted array from right to left and shifts these elements to the right until it finds the proper insert location.
- After all elements have been inserted, the array is sorted.

The pseudocode for the Insertion Sort is

```
for i = 1 to last array index by 1
    j = i
    temp = element at index i
    while (j != 0 and value of current element is less than value of element at index j - 1 )
        shift element at index j - 1 to the right
        decrement j by 1
assign current element value (stored in temp) to element at index j
```

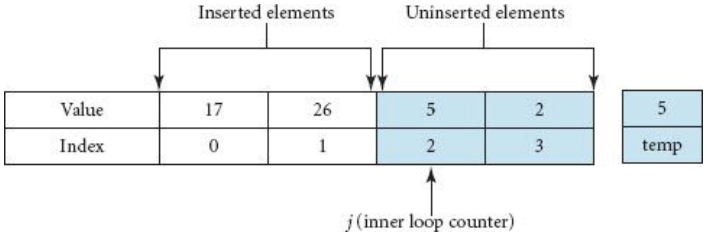
For example, let's walk through an Insertion Sort on the following array. At the beginning, the unsorted array is



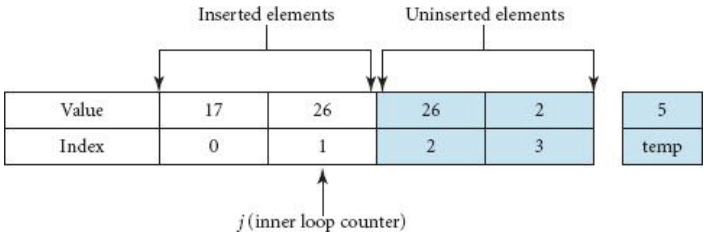
The first element of the array, 17, is automatically in the correct position when we consider the subarray as consisting of that element only. The value of the outer loop counter (*i*) is 1, and we will now insert the second array element, 26, into the left subarray. First, we save the value of the element to be inserted by storing it in *temp.* We need to save the value because it is possible that we will shift

other values, in which case we would overwrite that element. The value of the inner loop counter (*j*) is set to the value of the outer loop counter (*i*), i.e., 1. We compare elements 26 (index *j* = 1) and 17 (index *j* - 1 = 0). Since 26 is larger than 17, we exit the inner loop (and therefore we do not shift 17 to the right). We then assign the value of the current element, 26, stored in *temp,* to the element at index *j* = 1; in this case, there is no change to the array. The value 26 has been inserted.

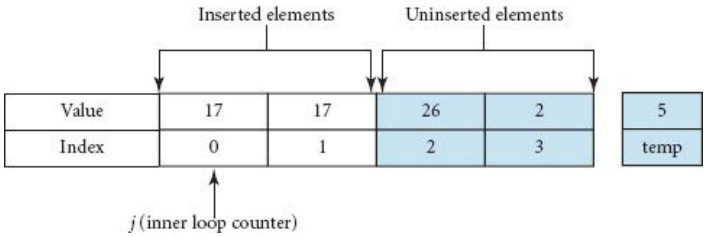
The outer loop counter (*i*) is incremented, and its value is 2. We will now insert the third array element, 5, into the left subarray (at this point comprised of the two inserted elements, 17 and 26).



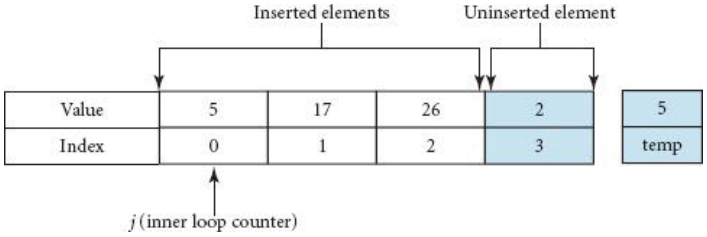
The value of the inner loop counter (*j)* is set to the value of the outer loop counter (*i*), i.e. 2. We compare the current element, 5, stored in *temp,* and 26 (index *j* - 1 = 1). Since 5 is smaller than 26, we shift 26 to the right and decrement *j* by 1; *j* now has the value 1.



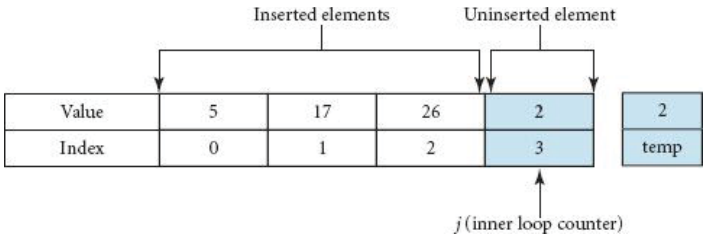
We then compare the current element, 5, stored in *temp,* and 17 (index *j* - 1 = 0). Since 5 is smaller than 17, we shift 17 to the right and decrement *j* by 1; *j* now has the value 0.



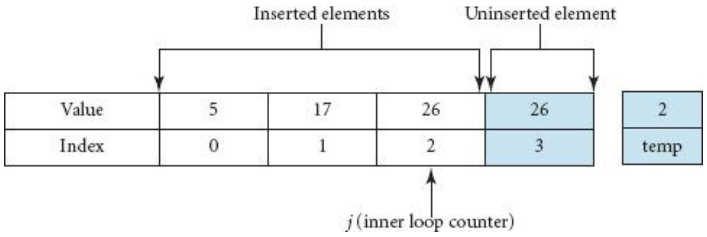
Since *j* is 0, we exit the inner loop and assign the value of the current element, 5, stored in *temp,* to the array element at index *j* = 0. The value 5 has now been inserted.



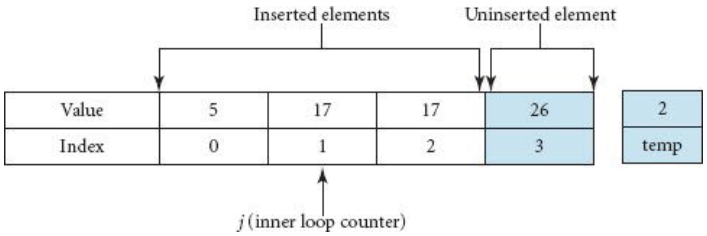
The outer loop counter (*i*) is incremented, and its value is 3. We will now insert the fourth array element, 2, into the left subarray (at this point comprised of the three inserted elements, 5, 17, and 26).



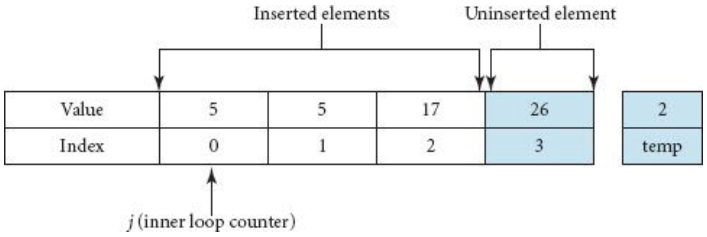
The value of the inner loop counter (*j)* is set to the value of the outer loop counter (*i*), i.e. 3. We compare the current element, 2, stored in *temp,* and 26 (index *j* - 1 = 2). Since 2 is smaller than 26, we shift 26 to the right and decrement *j* by 1; *j* now has the value 2.



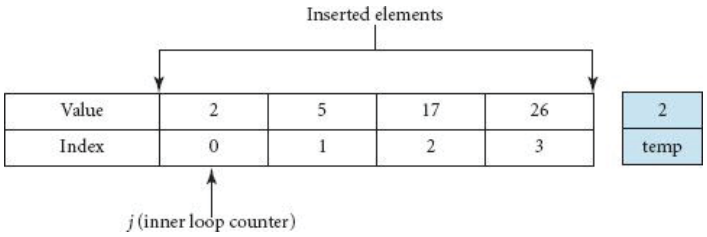
We then compare the current element, 2, stored in *temp,* and 17 (index *j* - 1 = 1). Since 2 is smaller than 17, we shift 17 to the right and decrement *j* by 1; *j* now has the value 1.



We then compare the current element, 2, stored in *temp,* and 5 (index *j* - 1 = 0). Since 2 is smaller than 5, we shift 5 to the right and decrement *j* by 1; *j* now has the value 0.



Since *j* is 0, we exit the inner loop and assign the value of the current element, 2, stored in *temp,* to the array element at index *j.* The value 2 has now been inserted.



The outer loop counter (*i*) is incremented, and its value is 4, which causes the outer loop to terminate. All the elements have been inserted; the array is now sorted.

Example 8.18 shows our *Sorter* class with the Insertion Sort algorithm implemented in lines 43-63.

COMMON ERROR TRAP

When looping through an array, be careful not to access an element outside the bounds of the array. Your code will compile, but will generate an *Array-IndexOutOfBounds-Exception* at run time.

```
 1 /* Sort Utility Class
 2    Anderson, Franceschi
 3 */
 4
 5 public class Sorter
 6 {
 7   /**  Performs a Selection Sort on
 8   *      an integer array
 9   *    @param the array to sort
10   */
11  public static void selectionSort( int [ ] array )
12  {
13     int temp; // temporary location for swap
14     int max;  // index of maximum value in subarray
15
16     for ( int i = 0; i < array.length; i++ )
17     {
18       // find index of largest value in subarray
```

```
      max = indexOfLargestElement( array, array.length - i );

      // swap array[max] and array[array.length - i - 1]
      temp = array[max];
      array[max] = array[array.length - i - 1];
      array[array.length - i - 1] = temp;
    }
 }

 /**  Finds index of largest element
 *    @param     size  the size of the subarray
 *    @return    the index of the largest element in the subarray
 */
 privatestatic int indexOfLargestElement( int [ ] array, int size )
 {
    int index = 0;
    for( int i = 1; i < size; i++ )
    {
        if ( array[i] > array[index] )
           index = i;
    }
    return index;
 }

 /**  Performs an Insertion Sort on an integer array
 *    @param array   array to sort
 */
 public static void insertionSort( int [ ] array )
 {
   int j, temp;

   for ( int i = 0; i < array.length; i++ )
   {
      j = i;
      temp = array[i];

      while ( j != 0 && array[j - 1] > temp )
      {
         array[j] = array[j - 1];
         j--;
      }

      array[j] = temp;
   }
 }
}
```

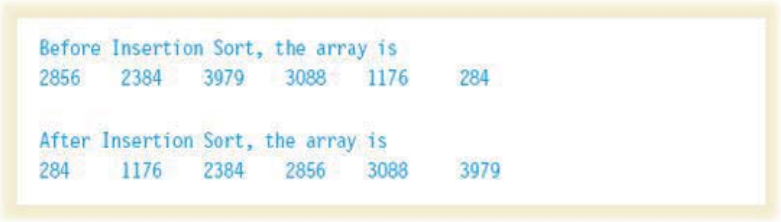
EXAMPLE 8.18 Sorter Class with Insertion Sort

Example 8.19 shows a client program that instantiates an integer array, fills it with random values, and then prints the array before and after performing the Insertion Sort. Figure 8.24 shows a sample run, using the Insertion Sort algorithm to sort an array of integers.

```
/** Client for Insertion Sort
*   Anderson, Franceschi
*/
import java.util.Random;

public class InsertionSortClient
{
  public static void main( String [ ] args )
  {
    // instantiate an array and fill with random values
    int [ ] numbers = new int [6];
    Random rand = new Random( );
    for ( int i = 0; i < numbers.length; i++ )
    {
      numbers[i] = rand.nextInt( 5000 ) + 1;
    }

    System.out.println( "Before Insertion Sort, the array is" );
    for ( int i = 0; i < numbers.length; i++ )
       System.out.print( numbers[i] + "\t" );
    System.out.println( );

    Sorter.insertionSort( numbers ); // sort the array

    System.out.println( "\nAfter Insertion Sort, the array is" );
    for ( int i = 0; i < numbers.length; i++ )
         System.out.print( numbers[i] + "\t" );
    System.out.println( );
  }
}
```

EXAMPLE 8.19 Using Insertion Sort

Figure 8.24 Using Insertion Sort



8.6.4 Sorting Arrays of Objects

We saw earlier in the chapter that data items to be sorted can be primitive data types, such as integers or *doubles.* But they can also be objects. With an array of objects, it is important to understand that we need to sort the objects themselves, not the array elements, which are merely the object references, or memory locations of the objects.

Arrays of objects are sorted using a sort key, which is one or more of the instance variables of the objects. For instance, if we have email objects, they can be sorted by date received, by author, by subject, and so on. It is important to note that when we sort objects, the integrity of the objects must be respected; for instance, when we sort a collection of email objects by sender, we sort a collection of email objects, not a collection of senders.

Thus, to perform the Insertion Sort on the *cars* array of *Auto* objects, we need to decide which field (or fields) of the *Auto* object determines the order of the objects. If we say that the *model* is the sort field, then the comparison statement would compare the models in two objects, that is, two *Strings.* As you recall from Chapter 5, the *compareTo* method of the *String* class compares the values of two *Strings.* It returns a positive number if the *String* for which the method is invoked is greater than the *String* passed as an argument.

To sort the *cars* array using an Insertion Sort, we would need to make several revisions to the *InsertionSort* method. First, the data type of the array must be declared as an *Auto* in the parameter list. Second, *temp* needs to be defined as an *Auto* reference, and finally, we need to substitute the *compareTo* method in the condition that compares array elements.

REFERENCE POINT

The *compareTo* method of the *String* class is discussed in detail in Chapter 5.

The revised Insertion Sort code becomes:

```
/* * Insertion sorts an array of Autos
*      @param arr an array of Autos
*/
public static void insertionSort( Auto [ ] arr )
{
  Auto temp;
  int j;

  for ( int i = 1; i < arr.length; i++ )

  {
     j = i;
     temp = arr[i];


     while ( j != 0 && ( temp.getModel( ) ).compareTo(
             arr[j - 1].getModel( ) )  < 0 )
     {
          arr[j] = arr[j - 1];
          j--;
     } // end while loop

     arr[j] = temp;

  } // end for loop
}  // end InsertionSort method
```

8.6.5 Sequential Search of a Sorted Array

Earlier in the chapter, the *DVDWinners* class sequentially searched an array. The algorithm assumed the elements were not in order. If we sort the array, a Sequential Search can be implemented more efficiently for the case when the search key is not present in the array. Instead of searching the entire array before discovering that the search key is not in the array, we can stop as soon as we pass the location where that element would be if it were in the array. In other words, if the array is sorted in ascending order, we can recognize an unsuccessful search when we find an element in the array that is greater than the search key. Because the array is

sorted in ascending order, all the elements after that array element are larger than that element, and therefore are also larger than the search key.

To implement this algorithm, we can add another test to the *for* loop condition, so that we exit the loop as soon as we find an element that is greater than the search key. The improved algorithm shown next could be used to replace the *indexOfWinner* method shown in Example 8.14 for Sequential Search of a sorted *winners* array:

```
publicint indexOfWinner( int key )
{
   for ( int i = 0; winners[i] <= key && i < winners.length; i++ )
   {
      if ( winners[i] == key )

      return i;
   }

   return -1; // end of array reached without finding key
              // or an element larger than the key was found
}
```

In fact, if the array is sorted, it can be searched even more efficiently using an algorithm called Binary Search, which we explain in the next section.

8.6.6 Binary Search of a Sorted Array

If you've played the “Guess a Number” game, you probably have used the concept of a **Binary Search.** In this game, someone asks you to guess a secret number between 1 and 100. For each number you guess, they tell you whether the secret number is larger or smaller than your guess. A good strategy is to guess the number in the middle, which in this example is 50. Whether the secret number is larger or smaller than 50, you will have eliminated half of the possible values. If the secret number is greater than 50, then you know your next guess should be 75 (halfway between 50 and 100). If the secret number is less than 50, your next guess should be 25 (halfway between 1 and 50). If you continue eliminating half the possible numbers with each guess, you will quickly guess the secret number. This approach works because we are “searching” a sorted set of numbers (1 to 100).

Similarly, a Binary Search of a sorted array works by eliminating half the remaining elements with each comparison. First, we look at the middle element of the array. If the value of that element is the search key, we return its index. If, however, the value of the middle element is greater than the search key, then the search key cannot be found in elements with array indexes higher than that element. Therefore, we will search the left half of the array only. Similarly, if the value of the middle element is lower than the search key, then the search key cannot be found in elements with array indexes lower than the middle element. Therefore, we will search in the right half of the array only. As we keep searching, the subarray we search keeps shrinking in size. In fact, the size of the subarray we search is cut in half at every iteration.

If the search key is not in the array, the subarray we search will eventually become empty. At that point, we know that we will not find our search key, and we return -1.

Example 8.20 shows our Binary Search algorithm.

```
 1  /** Binary Search
 2  *   Anderson, Franceschi
 3  */
 4
 5  import java.util.Scanner;
 6
 7  public class BinarySearcher
 8  {
 9    public static void main( String [ ] args )
10    {
11      // define an array sorted in ascending order
12      int [ ] numbers = { 3, 6, 7, 8, 12, 15, 22, 36, 45,
13                          48, 51, 53, 64, 69, 72, 89, 95 };
14
15      Scanner scan  = new Scanner( System.in );
16      System.out.print( "Enter a value to search for > " );
17      int key = scan.nextInt( );
18
19      int index = binarySearch( numbers, key );
20      if ( index != -1 )
21           System.out.println( key + " found at index " + index );
22      else
23           System.out.println( key + " not found" );
24    }
25
26    public static int binarySearch( int [ ] arr, int key )
27    {
28      int start = 0;
29      int end = arr.length - 1;
30      int middle;
31
32      while ( end >= start )
```

```
33      {
34        middle = ( start + end ) / 2; // element in middle of array
35
36        if ( arr[middle] == key )
37        {
38           return middle;       // key found at middle
39        }
40        else if ( arr[middle] > key )
41        {
42           end = middle - 1;   // search left side of array

43        }
44        else
45        {
46           start = middle + 1; // search right side of array
47        }
48      }
49      return -1;
50    }
51 }
```

EXAMPLE 8.20 Binary Search of a Sorted Array

We start by declaring and initializing an integer array with 17 sorted elements (lines 12-13). We then prompt the user for a search key and call the *binarySearch* method (lines 16-19).

The *binarySearch* method is coded at lines 26-50. The local variables *start* and *end* store the first and last index of the subarray to search. Because we begin by searching the entire array, we initialize these to the indexes of the first and last element of the array that was passed as a parameter. The local variable *middle,* declared at line 30, will store the index of the middle element in the subarray to search.

The search is performed in a *while* loop (lines 32-48), whose condition determines whether the subarray is empty. If the subarray is not empty, we calculate the value for *middle* by adding the indexes of the first and last elements and dividing by 2 (line 34). Next we test whether the value at the *middle* index is equal to the key. If so, we have found the key and we return its index, which is *middle* (lines 36-39). If not, we test whether the value in the middle of the subarray is greater than the key. If so, we reduce the subarray to the elements with indexes less than *middle* (lines 40-43) and greater than or equal to *start.* If the value in the middle of the subarray is less than the key, we reduce the subarray to the elements with indexes greater than *middle* (lines 44-47) and smaller than or equal to *end.*

When the *while* loop continues, we reevaluate the condition to determine whether the subarray is empty, and if not, continue making our comparisons and either returning the index of the search key or reducing the size of the subarray. If the search key is not in the array, the subarray eventually becomes empty, and we exit the *while* loop and return -1 (line 49). Figure 8.25 shows the output when the search key is found.

Figure 8.25 Output from Example 8.20

```
Enter a value to search for > 64
64 found at index 12
```

Let's run through the Binary Search algorithm on the key 7 to illustrate how the algorithm works when the key is found in the array. Here is the array *numbers*:

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

When the *binarySearch* method is called, it sets *start* to 0 and *end* to *arr.length -1,* which is 16. Thus, the value of *middle* is 8.

The element at index 8 (45) is greater than 7, so we set *end* to 7 (*middle - 1*), and we will now search the left subarray, highlighted next. The value of *middle* is now 3 ((0 + 7) / 2).

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The element at index 3 (8) is greater than 7, so we set *end* to 2 *(middle - 1)* and keep searching in the left subarray, highlighted next. The value of *middle* is now 1 ((0 + 2) / 2).

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The element at index 1 (6) is smaller than 7, so we set *start* to 2 *(middle + 1)* and search in the right subarray, highlighted next. The value of *middle* is now 2 ((2 + 2) / 2).

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The element at index 2 (7) is equal to 7. We have found the value and return its index, 2.

Let's now run the preceding example on the key 34 to illustrate how the algorithm works when the key is not found in the array.

Here is the array *numbers* again:

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Again, when the *binarySearch* method is called, it sets *start* to 0 and *end* to *arr.length - 1,* which is 16. Thus, *middle* is assigned the value 8 for the first comparison.

The element at index 8 (45) is greater than 34, so we set *end* to 7 (*middle- 1*), and keep searching in the left subarray. The value of *middle* becomes 3 for the next comparison.

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The element at index 3 (8) is smaller than 34, so we search in the right subarray highlighted below. The value of *middle* is now 5.

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The element at index 5 (15) is smaller than 34, so we search in the right subarray. The value of *middle* is now 6.

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

The element at index 6 (22) is smaller than 34, so we search in the right subarray. The value of *middle* is now 7.

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

At this point, *start, end,* and *middle* all have the value 7. The element at index 7 (36) is larger than 34, so we assign *end* the value *middle - 1,* which is 6. This makes *end* less than *start* and consequently makes the *while* loop condition evaluate to *false.* We have not found 34, so we return - 1.

8.7 Programming Activity 2: Searching and Sorting Arrays

In this activity, you will work again with a 15-element integer array, performing these activities:

1. Write a method to perform a Sequential Search of an array.
2. Write a method to implement the Bubble Sort algorithm to sort an array.

The basic approach to a Bubble Sort is to make multiple passes through the array. In each pass, we compare adjacent elements. If any two adjacent elements are out of order, we put them in order by swapping their values.

To sort an array of *n* elements in ascending order, Bubble Sort implements a double loop:

- The outer loop executes *n* - 1 times.

- For each iteration of the outer loop, the inner loop steps through all the unsorted elements of the array and does the following:
  - Compares the current element with the next element in the array.
  - If the next element is smaller, it swaps the two elements.

Outer loop counter	Indexes of element(s) at the sorted position
0	$n - 1$
1	$n - 2, n - 1$
2	$n - 3, n - 2, n - 1$
…	…
$n - 3$	$2, 3, 4, \ldots, n - 3, n - 2, n - 1$
$n - 2$	$1, 2, 3, 4, \ldots, n - 3, n - 2, n - 1$

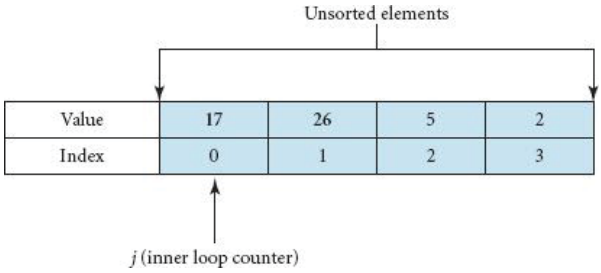
At this point, $n - 1$ elements have been moved to their correct positions. That leaves only the element at index 0, which is therefore automatically at the correct position within the array. The array is now sorted.

As the outer loop counter goes from 0 to $n - 2$, it iterates $n - 1$ times.

The pseudocode for the Bubble Sort is

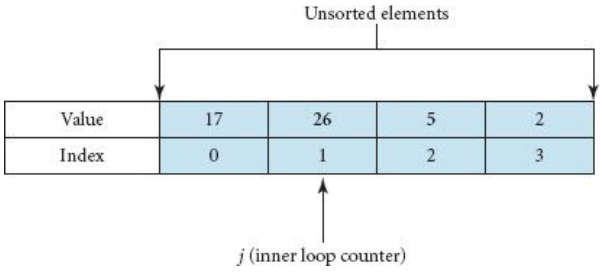
```
for i = 0 to last array index - 1 by 1
    for j = 0 to ( last array index - i -1 ) by 1
        if (2 consecutive elements are in the wrong order)
            swap them
```

For example, let's walk through a Bubble Sort on the following array. At the beginning, the unsorted array is

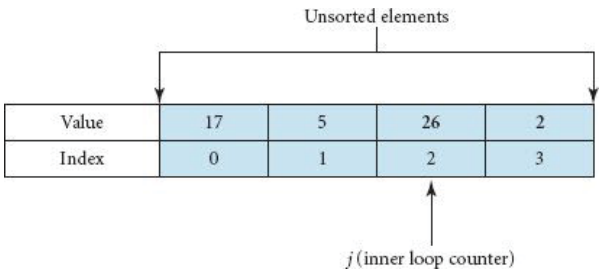


The value of the outer loop counter (i) is 0, and the value of the inner loop counter (j) is also 0. We compare elements 17 (index $j = 0$) and 26 (index $j + 1 = 1$). Since 17 is smaller than 26, we do not swap them.

The inner loop counter (j) is incremented, and its value is now 1.



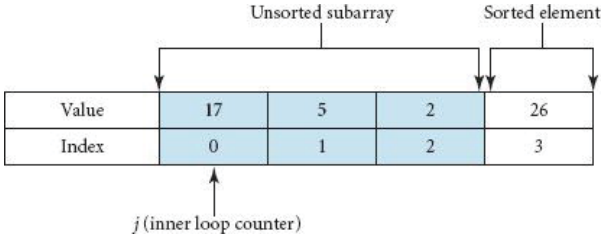
We compare elements 26 (index $j = 1$) and 5 (index $j + 1 = 2$). Since 26 is larger than 5, we swap them. The inner loop counter (j) is incremented, and its value is now 2.



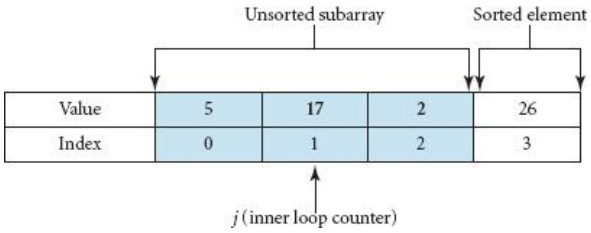
We compare elements 26 (index $j = 2$) and 2 (index $j + 1 = 3$). Since 26 is larger than 2, we swap them.

The inner loop counter (*j)* is incremented, and its value is now 3; therefore, we exit the inner loop. (We have reached the end of the unsorted sub-array, which at this point is the whole array.) At the end of one execution of the inner loop, the value 26 has "bubbled up" to its correct position within the array.

We now go back to the outer loop, and the outer loop counter (*i*) is incremented; its value is now 1. We reenter the inner loop and the value of the inner loop counter (*j)* is reinitialized to 0.



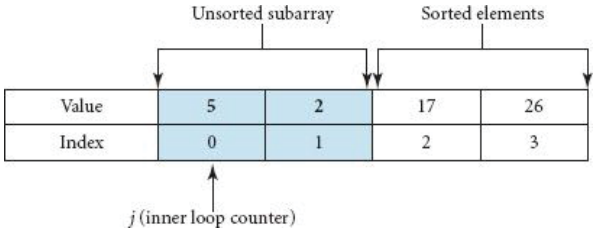
We compare elements 17 (index *j* = 0) and 5 (index *j* + 1 = 1). Since 17 is larger than 5, we swap them. The inner loop counter (*j*) is incremented, and its value is now 1.



We compare elements 17 (index *j* = 1) and 2 (index *j* + 1 = 2). Since 17 is larger than 2, we swap them.

The inner loop counter (*j*) is incremented, and its value is now 2; therefore, we exit the inner loop. (We have reached the end of the unsorted subarray.) At this point, the element 17 has "bubbled up" to its correct position within the array.

We go back to the outer loop, and the outer loop counter (*i*) is incremented; its value is now 2, and this will be the last iteration of the outer loop. We reenter the inner loop and the value of the inner loop counter (*j*) is reinitialized to 0.

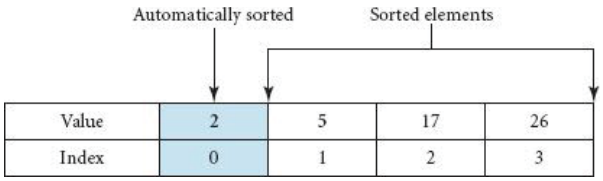


We compare elements 5 (index *j* = 0) and 2 (index *j* + 1 = 1). Since 2 is smaller than 5, we swap them.

The inner loop counter (*j)* is incremented, and its value is now 1; therefore, we exit the inner loop. (We have reached the end of the unsorted sub-array.) At this point, the element 5 has "bubbled up" to its correct position within the array.

We go back to the outer loop, and the outer loop counter (*i*) is incremented; its value is now 3, and therefore, we exit the outer loop. For the four elements in the array; we executed the outer loop three times.

The array is now sorted.



The framework for this Programming Activity will animate your algorithm so that you can watch your algorithm work and check the accuracy of your code. For example, Figure 8.26 demonstrates the Bubble Sort at work. At this point, the program has completed three passes through the array and is comparing the values of elements 5 and 6.

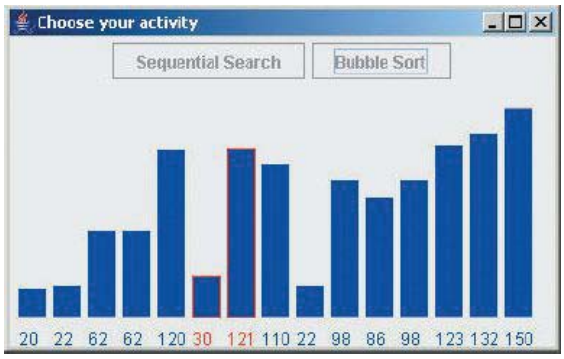
Instructions

In the Chapter 8 Programming Activity 2 directory on the CD-ROM accompanying this book, you will find the source files needed to complete this activity. Copy all the files to a directory on your computer. Note that all files should be in the same directory.

Open the *ArrayPractice2.java* source file. Searching for five stars (*****) in the source code will position you at the two locations where

you will add your code. Your first task is to complete the *sequentialSearch* method, which searches the *arr* array, an instance variable of the *ArrayPractice2* class. The array *arr* has already been instantiated for you and filled with random values. The second task is to complete the *bubbleSort* method. Example 8.21 shows the section of the *ArrayPractice2* source code where you will add your code. Note that in each method, you are asked to call the *animate* method so that your method code can be animated as it works. Note also that for the *sequentialSearch* method, we provide a dummy *return* statement *(return 0;)*. We do this so that the source code will compile. In this way, you can write and test each method separately, using stepwise refinement. When you are ready to write the *sequentialSearch* method, just replace the dummy *return* statement with the appropriate *return* statement for that method.

Figure 8.26 The Bubble Sort at Work




```
//  1 ***** student writes this method
/**  Searches for key in integer array named arr
//     arr is an instance variable of the class and has been
//     instantiated and filled with random values.
//  @param key value to search for
//  @return  if key is found, the  index of the first element
//     in array whose value is key; if key is not found,
//      the method returns -1
*/
public int sequentialSearch( int key )
{
// Note:  To animate the algorithm, put this method call as the
// first statement in your for loop
//  animate( i, 0 );
//         where i is the index of the current array element


   return 0; // replace this statement with your return statement

} // end of sequentialSearch

//  2. *****  student writes this method
/**  Sorts arr in ascending order using the bubble sort algorithm
*/
public void bubbleSort( )
{
 // Note:  To animate the algorithm, put this method call as the
 // last statement in your innermost for loop
 //  animate( i, j );
 //           where i is the value of the outer loop counter
 //            and j is the value of the inner loop counter,
 //            or the index of the current array element


} // end of bubbleSort
```

EXAMPLE 8.21 Student Section of *ArrayPractice2*

Figure 8.27 Opening Window of *ArrayPractice2*