CHAPTER 12

ARRAYS CONTINUED

OBJECTIVES

After completing this chapter, you will be able to:

- Write a method for searching an array
- Write a method for sorting an array
- Write methods to perform insertions and removals at given positions in an array
- Create and manipulate two-dimensional arrays

VOCABULARY

Binary search
Bubble sort
Insertion sort
Linear search
Multidimensional array
One-dimensional array
Ragged array
Selection sort
Two-dimensional array

In Chapter 9, we examined how to declare array variables, instantiate array objects, and manipulate arrays using the subscript operator, loops, and methods. This chapter covers more complex operations on arrays, such as searching, sorting, and the insertion and removal of elements. In addition, we explore how to create and manipulate two-dimensional arrays.

12.1 Searching

Searching collections of elements for a given target element is a very common operation in software systems. Some examples are the `indexOf` methods for strings discussed in Chapter 7. In this section, we examine two typical methods for searching an array of elements—a linear search and a binary search.

Linear Search

In Chapter 10, we developed the code for a method that searches an array of `int` for a given target value. The method returns the index of the first matching value or –1 if the value is not in the array. Following is the code:

```
int search (int[] a, int searchValue){
   for (int i = 0; i < a.length; i++)
      if (a[i] == searchValue)
         return i;
   return -1;
}
```

The method examines each element in sequence, starting with the first one, to determine if a target element is present. The loop breaks if the target is found. The method must examine every element to determine the absence of a target. This method of searching is usually called a ***linear search***.

Searching an Array of Objects

Suppose we have an array of names that we want to search for a given name. A name is a `String`. We cannot use the search method developed already. But we can use a similar loop in the code for a different search method that expects an array of `strings` and a target `string` as parameters. The only change in the loop is that two string elements must be compared with the method `equals` instead of the operator `==`. Following is the code:

```
int search (String[] a, String searchValue){
   for (int i = 0; i < a.length; i++)
      if (a[i].equals(searchValue))
         return i;
   return -1;
}
```

This method can be generalized to work with any object, not just strings. We simply substitute `Object` for `String` in the formal parameter list. The method still works for strings, and we can also use it to search an array of `Student` objects for a target student, assuming that the `Student` class includes an appropriate `equals` method. Following is a code segment that uses this single method to search arrays of two different types:

```
String[]  stringArray = {"Hi", "there", "Martin"};
Student[] studentArray = new Student[5];
Student   stu = new Student("Student 1");

for (int i = 0; i < studentArray.length; i++)
   studentArray[i] = new Student("Student " + (i + 1));

int stringPos = search(stringArray, "Martin");     // Returns 2
int studentPos = search(studentArray, stu);        // Returns 0
```

Binary Search

The linear search method works well for arrays that are fairly small (a few hundred elements). As the array gets very large (thousands or millions of elements), however, the behavior of the search degrades (more on this in Chapter 13). When we have an array of elements that are in ascending order, such as a list of numbers or names, there is a much better way to proceed. For example, in ordinary life, we do not use a linear search to find a name in a phone book. If we are looking for “Lambert,” we open the book at our estimate of the middle page. If we’re not on the “La” page, we look before or after, depending on whether we have opened the book at the “M” page or the “K” page. For computation, we can formalize this kind of search technique in an algorithm known as a ***binary search***. This method is much faster than a linear search for very large arrays.

The basic idea of a binary search is to examine the element at the array's midpoint on each pass through the search loop. If the current element matches the target, we return its position. If the current element is less than the target, then we search the part of the array to the right of the midpoint (containing the positions of the greater items). Otherwise, we search the part of the array to the left of the midpoint (containing the positions of the lesser items). On each pass through the loop, the current leftmost position or the current rightmost position is adjusted to track the portion of the array being searched. Following is a Java method that performs a binary search on an array of integers:

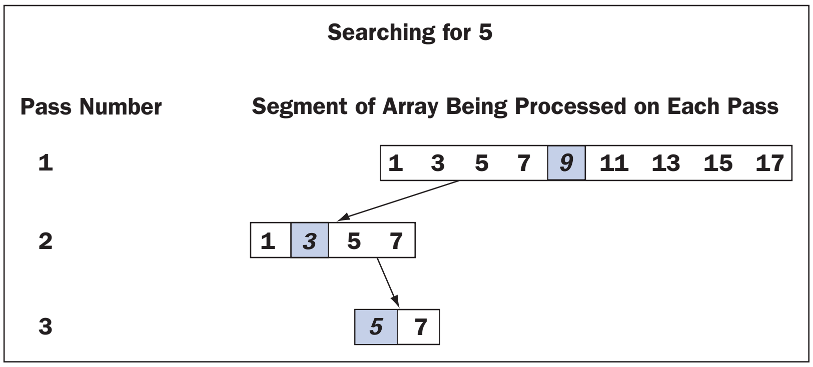
```
int search (int[] a, int searchValue){
   int left = 0;                           // Establish the initial
   int right = a.length - 1;               // endpoints of the array
   while (left <= right){                  // Loop until the endpoints cross
      int midpoint = (left + right) / 2;   // Compute the current midpoint
      if (a[midpoint] == searchValue)      // Target found; return its index
         return midpoint;
      else if (a[midpoint] < searchValue)  // Target to right of midpoint
         left = midpoint + 1;
      else                                 // Target to left of midpoint
         right = midpoint - 1;
   }
   return -1;                              // Target not found
}
```

Figure 12-1 shows a trace of a binary search for the target value 5 in the array

```
1 3 5 7 9 11 13 15 17
```

Note that on each pass through the loop, the number of elements yet to be examined is reduced by half. As we will see in more detail Chapter 13, herein lies the advantage of binary search over linear search for very large arrays.

FIGURE 12-1
A trace of a binary search of an array



Comparing Objects and the `Comparable` Interface

When using binary search with an array of objects, we must compare two objects. But objects do not understand the < and > operators, and we have seen that == is not a wise choice for comparing two objects for equality. However, classes that implement the `Comparable` interface include the method `compareTo`, which performs the three different comparisons. Here is the signature of `compareTo`:

```
public int compareTo(Object other)
```

The behavior of `compareTo` is summarized in Table 12-1.

TABLE 12-1
The behavior of method `compareTo`

USAGE OF `compareTo`	VALUE RETURNED
`obj1.compareTo(obj2)`	0 if `obj1` is equal to `obj2`, using equals
`obj1.compareTo(obj2)`	A negative integer, if `obj1` is less than `obj2`
`obj1.compareTo(obj2)`	A positive integer, if `obj1` is greater than `obj2`

For example, the `String` class implements the `Comparable` interface; thus, the second output of the following code segment is 0:

```
String str = "Mary";
System.out.println(str.compareTo("Suzanne"));    // Outputs -6
System.out.println(str.compareTo("Mary"));       // Outputs 0
System.out.println(str.compareTo("Bob"));        // Outputs 11
```

The other output integers are system dependent, but the first should be negative, whereas the third should be positive (in the example given, they are –6 and 11, respectively).

Before sending the `compareTo` message to an arbitrary object, that object must be cast to `Comparable`, because `Object` does not implement the `Comparable` interface or include a `compareTo` method. Following is the code for the binary search of an array of objects:

```
int search (Object[] a, Object searchValue){
   int left = 0;
   int right = a.length - 1;
   while (left <= right){
      int midpoint = (left + right) / 2;
      int result = ((Comparable)a[midpoint]).compareTo(searchValue);
      if (result == 0)
         return midpoint;
      else if (result < 0)
         left = midpoint + 1;
      else
         right = midpoint - 1;
   }
   return -1;
}
```

Implementing the Method compareTo

As mentioned earlier, objects that are ordered by the relations less than, greater than, or equal to must understand the `compareTo` message. Their class must implement the `Comparable` interface and their interface, if there is one, should also include the method `compareTo`. Suppose, for example, that the `Student` class of Chapter 6, which has no interface, is modified to support comparisons of students' names. Following are the required changes to the code:

```
public class Student implements Comparable{

   <data declarations>

   public int compareTo(Object other){

      // The parameter must be an instance of Student
      if (! (other instanceof Student))
         throw new IllegalArgumentException("Parameter must be a Student");

      // Obtain the student's name after casting the parameter
      String otherName = ((Student)other).getName();

      // Return the result of comparing the two students' names
      return name.compareTo(otherName);
   }

   <other methods>
}
```

EXERCISE 12.1

1. Why is a linear search called "linear"?

2. Write a linear search method that searches an array of objects for a target object.

3. Which elements are examined during a binary search of the array 34 56 78 85 99 for the target element 100?

4. Jack advises Jill of a modification to linear search that improves its performance when the array is sorted: If the target element is less than the current element, the target cannot be in the array. Modify the linear search method for integers to accomplish this.

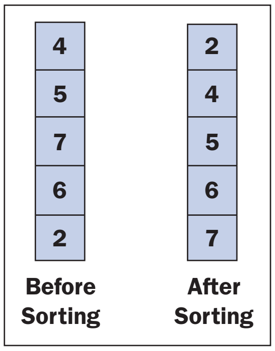
5. Describe what the following code segment does:

```
boolean inOrder = true;
for (int i = 0; i < a.length - 1; i++)
   if (a[i] > a[i + 1]){
      inOrder = false;
      break;
   }
```

12.2 Sorting

We have seen that if the elements in an array are in ascending order, we can write some efficient methods for searching the array. However, when the elements are in random order, we need to rearrange them before we can take advantage of any ordering. This process is called *sorting*. Suppose we have an array `a` of five integers that we want to sort from smallest to largest. In Figure 12-2, the values currently in `a` are as depicted on the left; we want to end up with values as they appear on the right.

FIGURE 12-2
An array before and after sorting



Many sort algorithms have been developed, and in this section we cover a few that are easy to write but not very efficient to run. More sophisticated and more efficient sort algorithms are discussed in Chapter 13.

Selection Sort

The basic idea of a ***selection sort*** is as follows:

```
For each index position i
   Find the smallest data value in the array from positions i
      through length - 1, where length is the number of data values stored.
   Exchange the smallest value with the value at position i
```

Table 12-2 shows a trace of the elements of an array after each exchange of elements is made. The items just swapped are marked with asterisks, and the sorted portion is shaded. Notice that in the second and fourth passes, because the current smallest numbers are already in place, we need not exchange anything. Also, after the last exchange, the number at the end of the array is automatically in its proper place.

TABLE 12-2
A trace of the data during a selection sort

UNSORTED ARRAY	AFTER 1ST PASS	AFTER 2ND PASS	AFTER 3RD PASS	AFTER 4TH PASS
4	1*	1	1	1
2	2	2*	2	2
5	5	5	3*	3
1	4*	4	4	4*
3	3	3	5*	5

Before writing the algorithm for this sorting method, note the following:

- If the array is of length n, we need $n - 1$ steps.
- We must be able to find the smallest number.
- We need to exchange appropriate array items.

When the code is written for this sort, note that strict inequality (<) rather than weak inequality (<=) is used when looking for the smallest remaining value. The algorithm to sort by selection is

```
For each i from 0 to n - 1 do
   Find the smallest value among a[i], a[i + 1], . . . a[n - 1]
   and store the index of the smallest value in minIndex
   Exchange the values of a[i] and a[index], if necessary
```

In Chapter 10, we saw a segment of the code we need to find the smallest value of array `a`. With suitable changes, we will incorporate this segment of code in a method, `findMinimum`, for the selection sort. We also will use a method `swap` to exchange two elements in an array.

Using these two methods, the implementation of a selection sort method is

```
void selectionSort(int[] a){
   for (int i = 0; i < a.length - 1; i++){
      int minIndex = findMinimum(a, i);
      if (minIndex != i)
         swap(a, i, minIndex);
   }
}
```

The method for finding the minimum value in an array takes two parameters, the array and the position, to start the search. The method returns the index position of the minimum element in the array. Its implementation uses a `for` loop:

```
int findMinimum(int[] a, int first){
   int minIndex = first;
```

```
      for (int i = first + 1; i < a.length; i++)
         if (a[i] < a[minIndex])
            minIndex = i;

      return minIndex;
   }
```

The swap method exchanges the values of two array cells:

```
   void swap(int[] a, int x, int y){
      int temp = a[x];
      a[x] = a[y];
      a[y] = temp;
   }
```

Bubble Sort

Given a list of items stored in an array, a ***bubble sort*** causes a pass through the array to compare adjacent pairs of items. Whenever two items are out of order with respect to each other, they are swapped. The effect of such a pass through an array of items is traced in Table 12-3. The items just swapped are marked with asterisks, and the sorted portion is shaded. Notice that after such a pass, we are assured that the array will have the item that comes last in order in the final array position. That is, the last item will "sink" to the bottom of the array, and preceding items will gradually "percolate" to the top.

TABLE 12-3
A trace of the data during one pass of a bubble sort

UNSORTED ARRAY	AFTER 1ST SWAP	AFTER 2ND SWAP	AFTER 3RD SWAP	AFTER 4TH SWAP
5	4*	4	4	4
4	5*	2*	2	2
2	2	5*	1*	1
1	1	1	5*	3*
3	3	3	3	5*

The bubble sort algorithm involves a nested loop structure. The outer loop controls the number of (successively smaller) passes through the array. The inner loop controls the pairs of adjacent items being compared. If we ever make a complete pass through the inner loop without having to make an interchange, we can declare the array sorted and avoid all future passes through the array. A pseudocode algorithm for bubble sort is

```
Initialize counter k to zero
Initialize boolean exchangeMade to true
While (k < n - 1) and exchangeMade
   Set exchangeMade to false
   Increment counter k
   For each j from 0 to n - k
      If item in jth position > item in (j + 1)st position
         Swap these items
         Set exchangeMade to true
```

A complete Java method to implement a bubble sort for an array of integers is shown in the following code:

```
void bubbleSort(int[] a){
   int k = 0;
   boolean exchangeMade = true;

   // Make up to n - 1 passes through array, exit early if no exchanges
   // are made on previous pass

   while ((k < a.length - 1) && exchangeMade){
      exchangeMade = false;
      k++;
      for (int j = 0; j < a.length - k; j++)
         if (a[j] > a[j + 1]){
            swap(a, j, j + 1);
            exchangeMade = true;
         }
      }
}
```

Insertion Sort

Although it reduces the number of data interchanges, the selection sort apparently will not allow an effective—and automatic—loop exit if the array becomes ordered during an early pass. In this regard, bubble sort is more efficient than selection sort for an array that is nearly ordered from the beginning. Even with just one item out of order, however, bubble sort's early loop exit can fail to reduce the number of comparisons that are made.

The ***insertion sort*** attempts to take greater advantage of an array's partial ordering. The goal is that on the *k*th pass through, the *k*th item among

```
a[0], a[1], ..., a[k]
```

should be inserted into its rightful place among the first *k* items in the array. Thus, after the *k*th pass (*k* starting at 1), the first *k* items of the array should be in sorted order. This is analogous to the fashion in which many people pick up playing cards and order them in their hands. Holding the first (*k* – 1) cards in order, a person will pick up the *k*th card and compare it with cards already held until its appropriate spot is found. The following steps will achieve this logic:

```
For each k from 1 to n - 1 (k is the index of array element to insert)
   Set itemToInsert to a[k]
   Set j to k - 1
   (j starts at k - 1 and is decremented until insertion position is found)
    While (insertion position not found) and (not beginning of array)
       If itemToInsert < a[j]
          Move a[j] to index position j + 1
```

```
            Decrement j by 1
         Else
            The insertion position has been found
            itemToInsert should be positioned at index j + 1
```

In effect, for each pass, the index *j* begins at the (*k* – 1)st item and moves that item to position *j* + 1 until we find the insertion point for what was originally the *k*th item.

An insertion sort for each value of *k* is traced in Table 12-4. In each column of this table, the data items are sorted in order relative to each other above the item with the asterisk; below this item, the data are not affected.

TABLE 12-4
A trace of the data during an insertion sort

UNSORTED ARRAY	AFTER 1ST PASS	AFTER 2ND PASS	AFTER 3RD PASS	AFTER 4TH PASS
2	2	1*	1	1
5 ←	5 (no insertion)	2	2	2
1	1 ←	5	4*	3*
4	4	4 ←	5	4
3	3	3	3 ←	5

To implement the insertion sort algorithm in Java, we use the following code:

```java
void insertionSort(int[] a){
   int itemToInsert, j;
   boolean stillLooking;

   // On the kth pass, insert item k into its correct position among
   // the first k entries in array.

   for (int k = 1; k < a.length; k++){
      // Walk backward through list, looking for slot to insert a[k]
      itemToInsert = a[k];
      j = k - 1;
      stillLooking = true;

      while ((j >= 0) && stillLooking )
         if (itemToInsert  < a[j]) {
            a[j + 1] = a[j];
            j--;
         }else
            stillLooking = false;
         // Upon leaving loop, j + 1 is the index
         // where itemToInsert  belongs
         a[j + 1] = itemToInsert;
   }
}
```

Sorting Arrays of Objects

Any of the sort methods can be modified to sort arrays of objects. We assume that the objects implement the `Comparable` interface and support the method `compareTo`. Then, we simply replace the element type of all array parameters with `Object` and make the appropriate use of `compareTo` in which the comparison operators are used. For example, here is the relevant change for the selection sort in the method `findMinimum`:

```
int findMinimum(Object[] a, int first){
   int minIndex = first;

   for (int i = first + 1; i < a.length(); i++)
      if (((Comparable)a[i]).compareTo(a[minIndex]) < 0)
         minIndex = i;

   return minIndex;
}
```

Testing Sort Algorithms

The sort algorithms developed thus far should be tested. The skeleton of a short tester program follows. The program loads an array with 20 random integers between 0 and 99, displays the array's contents, runs a sort method, and displays the array's contents again. Each sort method and its helper methods should be defined as `private static`.

```
// Example 12.1: Test sort algorithms

import java.util.Random;

public class TestSortAlgorithms{

   public static void main(String[] args){
      Random gen = new Random();
      int[] a = new int[20];

      //Initialize the array to random numbers between 0 and 99
      for (int i = 0; i < a.length; i++)
         a[i] = gen.nextInt(100);

      printArray(a);
      selectionSort(a);             // Pick one of three to test
      //bubbleSort(a);
      //insertionSort(a);
      printArray(a);
   }

   private static void printArray(int[] a){
      for (int i : a)
         System.out.print(i  + " ");
```

```
            System.out.println("");
        }

        // private static sort methods and their helpers go here
    }
```

You also should test the methods with an array that is already sorted.

EXERCISE 12.2

1. Draw a diagram that shows the contents of the array 8 7 6 5 4 after each number is moved in a selection sort.
2. Draw a diagram that shows the contents of the array 8 7 6 5 4 after each number is moved in a bubble sort, until the 8 arrives at the end of the array.
3. Describe the behavior of the selection sort, bubble sort, and insertion sort with an array that is already sorted. How many exchanges are made in each sort for an array of size *n*?
4. Modify the bubble sort method so that it sorts an array of objects.

12.3 Insertions and Removals

In Chapter 10, we discussed how to add or remove an element at the end of an array that is not full. In this section, we show how to perform these operations at arbitrary positions within an array. For simplicity, we make four assumptions:

1. Arrays are of fixed size; thus, when an array becomes full, insertions are not performed.
2. We are working with an array of objects, although we can modify the code to cover arrays of integers, employees, or whatever element type is desired.
3. For successful insertions, 0 <= target index <= logical size. The new element is inserted before the element currently at the target index, or after the last element if the target index equals the logical size.
4. For successful removals, 0 <= target index < logical size.

When an assumption is not satisfied, the operation is not performed and we return `false`; otherwise, the operation is performed and we return `true`.

In the code segments that follow, we use the following data declarations:

```
final int DEFAULT_CAPACITY = 5;
int logicalSize = 0;
Object[] array = new Object[DEFAULT_CAPACITY];
```

As you can see, the array has an initial logical size of 0 and a default physical capacity of 5. For each operation that uses this array, we provide a description of the implementation strategy and an annotated Java code segment. At the end of this section, we ask you to develop some static methods to perform these operations on arrays.

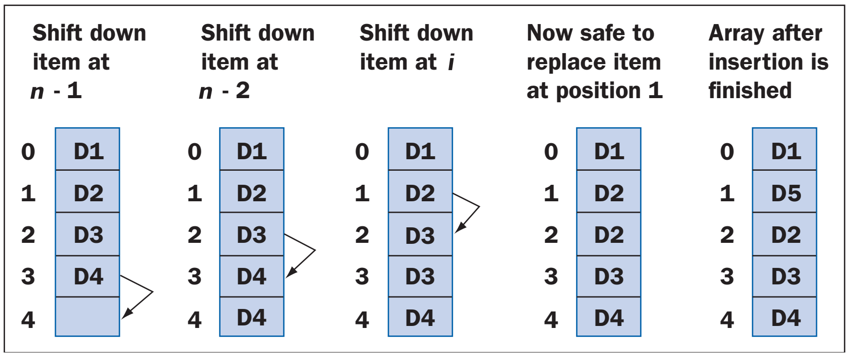
Inserting an Item into an Array at an Arbitrary Position

Inserting an item into an array differs from replacing an item in an array. In the case of a replacement, an item already exists at the given index position and a simple assignment suffices. Moreover, the logical size of the array does not change. In the case of an insertion, we must do six things:

1. Check for available space before attempting an insertion; if there is no space, return `false`.
2. Check the validity of the target index and return `false` if it is not >= 0 and <= logical size.
3. Shift the items from the logical end of the array to the target index down by one position.
4. Assign the new item to the cell at the target index.
5. Increment the logical size by one.
6. Return true.

Figure 12-3 shows these steps for the insertion of an item at position 1 in an array of four items.

FIGURE 12-3
Inserting an item into an array



As you can see, the order in which the items are shifted is critical. If we had started at the target index and copied down from there, we would have lost two items. Thus, we must start at the logical end of the array and work back up to the target index, copying each item to the cell of its successor. Following is the Java code for the insertion operation:

```java
// Check for a full array and return false if full
if (logicalSize == array.length)
   return false;

// Check for valid target index and return false if not valid
if (targetIndex < 0 || targetIndex > logicalSize)
   return false;

// Shift items down by one position
for (int i = logicalSize; i > targetIndex; i--)
   array[i] = array[i - 1];
```

```
// Add new item, increment logical size, and return true
array[targetIndex] = newItem;
logicalSize++;
return true;
```

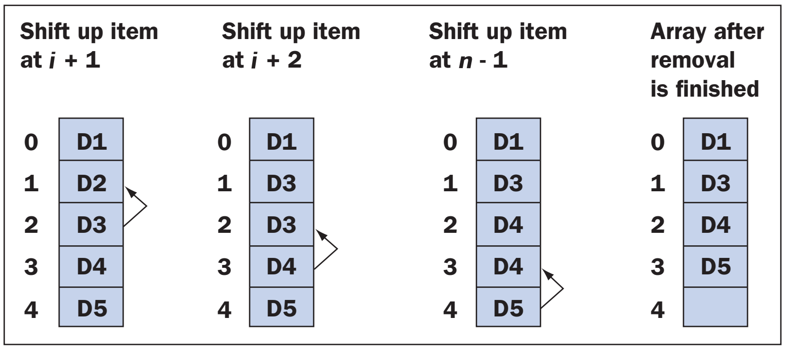
Removing an Item from an Array

Removing an item from an array involves the inverse process of inserting an item into the array. Following are the steps in this process:

1. Check the validity of the target index and return `false` if it is not >= 0 and < logical size.
2. Shift the items from the target index to the logical end of the array up by one position.
3. Decrement the logical size by one.
4. Return true.

Figure 12-4 shows these steps for the removal of an item at position 1 in an array of five items.

FIGURE 12-4
Removing an item from an array



As with insertions, the order in which we shift items is critical. For a removal, we begin at the item following the target position and move toward the logical end of the array, copying each item to the cell of its predecessor. Following is the Java code for the removal operation:

```
// Check for valid target index and return false if not valid
if (targetIndex < 0 || targetIndex >= logicalSize)
   return false;

// Shift items up by one position
for (int i = targetIndex; i < logicalSize - 1; i++)
   array[i] = array[i + 1];

// Decrement logical size and return true
logicalSize--;
return true;
```

Of course, there are other ways to handle potential errors in these operations, as discussed in earlier chapters. Instead of returning a Boolean value, one can return a string indicating the type of error or `null` indicating success. Alternatively, one can throw an exception for each error and return `void`.

A Tester Program for Array Methods

The operations just discussed are so frequently used that it is a good idea to provide methods for them. Ideally, one implements them as `static` methods in a class that serves a utility function similar to Java's `Math` class. In the following code, we specify two of these methods in the context of a tester program. We leave their complete development for you to try in Exercise 12.3, Question 2.

A method to insert a new item at a given index position, `insertItem`, expects the array, its logical size, the target index, and the new item as parameters. `insertItem` returns `true` if the operation is successful and `false` otherwise. This method does not increment the logical size; that responsibility is left to the client, who must check the Boolean value returned to take the appropriate action. A similar method named `removeItem` can be developed for removals.

Following is a short tester program that uses the method described previously:

```
// Example 12.2: Test insertions and removals

public class TestInsertAndRemove{

   public static void main(String[] args){

      // Create an initial array with 3 positions
      String[] array = new String[3];
      int logicalSize = 0;
      boolean successful = false;

      // Insert strings at positions 0, 1, 1, and 0
      successful = insertItem(array, logicalSize, 0, "Jack");
      if (successful)
         logicalSize++;

      successful = insertItem(array, logicalSize, 1, "Jill");
      if (successful)
         logicalSize++;

      successful = insertItem(array, logicalSize, 1, "sees");
      if (successful)
         logicalSize++;

      successful = insertItem(array, logicalSize, 0, "Before");
      if (successful)
         logicalSize++;

      // Display new logical size and contents
      System.out.println(logicalSize);
      for (int i = 0; i < logicalSize; i++)
         System.out.print(array[i] + " ");
   }

   // Definitions of array methods go here
   private static boolean insertItem(Object[] array, int logicalSize,
                                     int targetIndex, Object newItem){
      // Exercise
   }
   private static boolean removeItem(Object[] array, int logicalSize,
                                     int targetIndex){
      // Exercise
   }

}
```

Although our methods allow clients to perform insertions and removals, clients must still track the logical size of an array and update it. In addition, there is an upper bound on the number of items a client can insert into an array. We will learn how to overcome these limitations of arrays in Chapter 14.

EXERCISE 12.3

1. Describe the design strategy for inserting an element at an arbitrary position in an array.
2. Complete the two static methods for insertion and removal of an element at an arbitrary position in an array.
3. Describe what the following code segments do:

   a.

```
if (logicalSize == a.length){
   int[] temp = new int[a.length * 2];
   for (int i = 0; i < a.length; i++)
      temp[i] = a[i];
   a = temp;
}
```

   b.

```
if (a.length >= logicalSize * 4){
   int[] temp = new int[a.length * 2];
   for (int i = 0; i < a.length; i++)
      temp[i] = a[i];
   a = temp;
}
```

12.4 Two-Dimensional Arrays

The arrays we have been studying so far can represent only simple lists of items and are called ***one-dimensional arrays***. For many applications, ***multidimensional arrays*** are more useful. A table of numbers, for instance, can be implemented as a ***two-dimensional array***. Figure 12-5 shows a two-dimensional array with four rows and five columns.

FIGURE 12-5
A two-dimensional array with four rows and five columns

	col 0	col 1	col 2	col 3	col 4
row 0	00	01	02	03	04
row 1	10	11	12	13	14
row 2	20	21	22	23	24
row 3	30	31	32	33	34

Suppose we call the array `table`; then to indicate an element in `table`, we specify its row and column position, remembering that indexes start at 0:

```
x = table[2][3];  // Set x to 23, the value in (row 2, column 3)
```

Sum the Elements

The techniques for manipulating one-dimensional arrays are easily extended to two-dimensional arrays. For instance, the following is code that sums all the numbers in `table`. The outer loop iterates four times and moves down the rows. Each time through the outer loop, the inner loop iterates five times and moves across a different row.

```
int sum = 0;
for (int i = 0; i < 4; i++){     // There are four rows: i = 0,1,2,3
   for (int j = 0; j < 5; j++){ // There are five columns: j = 0,1,2,3,4
      sum += table[i][j];
   }
}
```

This segment of code can be rewritten without using the numbers 4 and 5. The value `table.length` equals the number of rows, and `table[i].length` is the number of columns in row `i`.

```
int sum = 0;
for (int i = 0; i < table.length; i++){
   for (int j = 0; j < table[i].length; j++){
      sum += table[i][j];
   }
}
```

Sum the Rows

Rather than accumulate all the numbers into a single sum, we now compute the sum of each row separately and place the results in a one-dimensional array called `rowSum`. This array has four elements, one for each row of the table. The elements in `rowSum` are initialized to 0 automatically by virtue of the declaration.

```
int[] rowSum = new int[4];
for (int i = 0; i < table.length; i++){
   for (int j = 0; j < table[i].length; j++){
      rowSum[i] += table[i][j];
   }
}
```

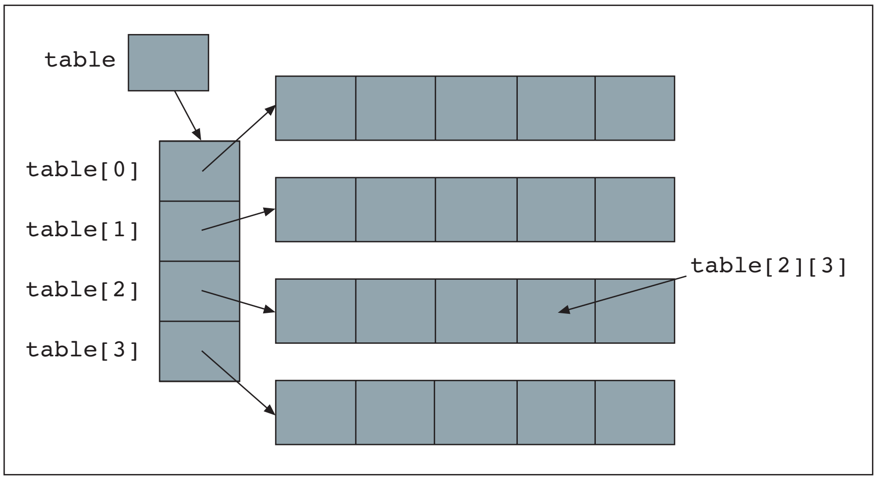
Declare and Instantiate

Declaring and instantiating two-dimensional arrays is accomplished by extending the processes used for one-dimensional arrays:

```
int[][] table;             // The variable table can reference a
                           // two-dimensional array of integers.
table = new int[4][5];     // Instantiate table as an array of size 4,
                           // each of whose elements will reference an array
                           // of 5 integers.
```

Figure 12-6 shows another diagram of `table` that illustrates the perspective revealed in the previous piece of code. The variable `table` references an array of four elements. Each of these elements in turn references an array of five integers. Although the diagram is complex, specifying an element in the resulting two-dimensional array is the same as before, for instance, `table[2][3]`.

FIGURE 12-6
Another way of visualizing a two-dimensional array



Initializer lists can be used with two-dimensional arrays. This requires a list of lists. The number of inner lists determines the number of rows, and the size of each inner list determines the size of the corresponding row. The rows do not have to be the same size, but they are in this example:

```
int[][] table = {{ 0, 1, 2, 3, 4},     // row 0
                 {10,11,12,13,14},     // row 1
                 {20,21,22,23,24},     // row 2
                 {30,31,32,33,34}};    // row 3
```

Variable Length Rows

Occasionally, the rows of a two-dimensional array are not all the same length. We call these ***ragged arrays***, and we just mention them in passing. Consider the following improbable declaration:

```
int[][] table;
table    = new int[4][];   // table has 4 rows
table[0] = new int[6];     // row 0 has 6   elements
table[1] = new int[10];    // row 1 has 10  elements
table[2] = new int[100];   // row 2 has 100 elements
table[3] = new int[1];     // row 3 has 1   element
```

Finally, remember that all the elements of a two-dimensional array must be of the same type, whether they are integers, doubles, strings, or whatever.

EXERCISE 12.4

1. What are two-dimensional arrays?
2. Write a code segment that declares a variable to reference an array of integers with 10 rows and 20 columns and assigns this variable a new array object.
3. Write a code segment that searches a two-dimensional array for a negative integer. The loop should terminate at the first instance of a negative integer in the array, and the variables `row` and `col` should be set to its position. Otherwise, if there are no negative integers in the array, the variables `row` and `col` should equal the number of rows and columns in the array (we assume that each row has the same number of columns).
4. Describe the contents of the array after the following code segment is run:

```
int [][] matrix = new int[5][5];

for (int row = 0; row < matrix.length; row++)
   for (int col = 0; col < matrix[row].length; col++)
      matrix[row][col] = row * col;
```

5. Write a code segment that outputs the integers in a two-dimensional array named `table`. The output should begin each row of integers on a new line.

12.5 Applications of Two-Dimensional Arrays

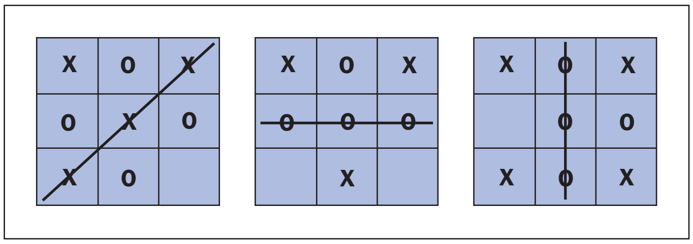
Two-dimensional arrays are most useful for representing information in a two-dimensional grid. For example, the players of the games checkers and chess move pieces around on a two-dimensional board of painted squares. Each square or position on a game board is located by specifying a row and a column. Graphics applications represent images as a set of pixels or color values located in a two-dimensional grid, using a screen coordinate system.

In this section, we examine two applications of two-dimensional arrays: a simple database for tracking golf scores and a tic-tac-toe game.

The Game of Tic-Tac-Toe

Tic-tac-toe uses a 3 by 3 grid of squares that are initially empty. Two players take turns by placing an X or an O in the empty squares, until one player is able to draw a horizontal, vertical, or diagonal line through three squares containing Xs or Os. Figure 12-7 shows several winning configurations.

FIGURE 12-7
Tic-tac-toe boards



The game can be played between a human user and the computer or between two human users. For now, we explore a two-person game, and leave the development of a computer-person game an exercise for you.

Let's think of the game board as an object that allows the user to

- View the state of the game in a two-dimensional layout
- Attempt to place an X or an O at a given position, with a signal of success or failure to do so
- Determine if a game has been won, and if so, by whom (either the owner of the Xs or the owner of the Os)
- Determine if the board is full, having no more empty positions (if this is the case, the outcome of the game might be a tie)
- Reset the board to start a new game

Given this analysis of the required behavior, we can now specify a set of methods in the interface of a new Java class called `TTTBoard`. Table 12-5 lists these methods and explains what they do.

TABLE 12-5
The methods of the `TTTBoard` class

`TTTBoard` METHOD	WHAT IT DOES
`TTTBoard()`	Creates a tic-tac-toe board whose cells initially contain hyphens.
`String toString()`	Returns a string representation of the board.
`void reset()`	Places hyphens in all of the board's cells.
`boolean placeXorO(char player, int row, int column)`	Preconditions: `player` must be `'X'` or `'O'`, 1 <= `row` <= 3 and 1 <= `column` <= 3. Attempts to place the character at the given row and column in the board. Returns `true` if placement was successful or `false` otherwise.
`char getWinner()`	Returns `'-'` if there is no winner yet. Otherwise, returns `'X'` or `'O'` to indicate the identity of the winner.
`boolean full()`	Returns `true` if the board contains only Xs and Os or `false` otherwise.

Assuming that you will eventually implement the `TTTBoard` class, you can now write a driver program that allows two users to play the game interactively (at the same computer). Here is the code for this program:

```
// Example 12.3

import java.util.Scanner;
import java.util.Random;

public class PlayTTT{

   public static void main(String[]args){
      // Instantiate a keyboard scanner and a board
      Scanner reader = new Scanner(System.in);
      TTTBoard board = new TTTBoard();

      // Display the empty board
      System.out.println(board);

      // Randomly decide who goes first
      Random gen = new Random();
      char player;
      if (gen.nextInt(2) == 1)
         player = 'O';
      else
         player = 'X';

      // Loop while there is no winner and the board is not full
      while (board.getWinner() == '-' && !board.full()){
         // Prompt the user for a move
         System.out.println(player + "'s turn");
         System.out.print("Enter the row and column[1-3, space, 1-3]: ");

         // Read the move
         int row = reader.nextInt();
         int column = reader.nextInt();

         // Attempt the move
         // If the move is illegal
         //    display an error message
         // Else
         //    display the board and switch players
         boolean success = board.placeXorO(player, row, column);
         if (! success)
            System.out.println("Error: cell already occupied!");
         else{
            System.out.println(board);
            if (player == 'X')
               player = 'O';
            else
               player = 'X';
         }
      }

      // Display results
      char winner = board.getWinner();
      if (winner != '-')
         System.out.println(winner + "s win!");
      else
         System.out.println("It's a draw!");
   }
}
```

Here is a transcript of the last two moves of a game played with this program:

```
X's turn
Enter the row and column[1-3, space, 1-3]: 3 2
O X -
X O O
O X X

O's turn
Enter the row and column[1-3, space, 1-3]: 1 3
O X O
X O O
O X X

Os win!
```

Within the `TTTBoard` class, the board can be represented as a 3 by 3 array of characters. After the instantiation of `TTTBoard`, the array's cells contain hyphens. The Xs and Os are represented by the characters 'X' and 'O', respectively. The methods for setting up the board, modifying a cell, and building a string representation are familiar from your study of arrays. Here is the code for these methods:

```
// Example 12.3 (continued)

public class TTTBoard{

   private char[][] board;

   public TTTBoard(){
      board = new char[3][3];
      reset();
   }

   public void reset(){
      for (int row = 0; row < 3; row++)
         for (int column = 0; column < 3; column++)
            board[row][column] = '-';
   }

   public String toString(){
      String result = "\n";
      for (int row = 0; row < 3; row++){
         for (int column = 0; column < 3; column++)
            result += board[row][column] + " ";
         result += "\n";
      }
      return result;
   }

   public boolean placeXorO(char player, int row, int column){
      if (board[row - 1][column - 1] == '-'){
         board[row - 1][column - 1] = player;
         return true;
      }
      else
         return false;
   }

   // Methods to test for fullness and to check for a winner
   // go here
}
```

There are various strategies to check the board for a winner. One way is to traverse all of the rows, columns, and diagonals in the board and build an array of strings from sequences of characters contained there. You can then search this array for a match to a three-letter string built from the ‘X’ or the ‘O’. The implementation of this method is left as an exercise for you.

Tracking Golf Scores

In the game of golf, a player typically plays a round of 18 holes in one day and records a numeric score for each hole. A player in a weekly league might be interested in tracking her scores over several rounds and analyzing the results. Example results are the lowest day, the highest day, the lowest average hole, and the highest average hole. Let's develop a program that inputs scores from a file and outputs these results.

The format of the input file contains the number of rounds on the first line. Each line following contains 19 numbers, the first one representing the date of the round and the remaining ones representing the scores for that round's holes.

At start-up, the program prompts the user for the name of the input file, or "Q" to quit the program. The program reads the information from the file and outputs it, together with the results of the analysis mentioned earlier. Figure 12-8 shows a sample session with this program.

FIGURE 12-8
Sample session of golf program

```
Enter the name of a scores file or Q to quit: scores.txt

Here is the complete history of scores:
Date: 20090601 Scores: 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
Date: 20090615 Scores: 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
Date: 20090630 Scores: 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
Date: 20090715 Scores: 3 3 3 3 4 4 4 4 5 5 5 5 4 4 4 4 4 4

The best day:  20090630 score: 54
The worst day: 20090601 score: 90

The best hole:  1 average: 3.75
The worst hole: 9 average: 4.25

Enter the name of a scores file or Q to quit: q
```

The class `GolfScoreCard` represents a scorecard for the program. This class also does the work of inputting a file and returning the results of analysis upon request. Table 12-6 shows the interface for this class.

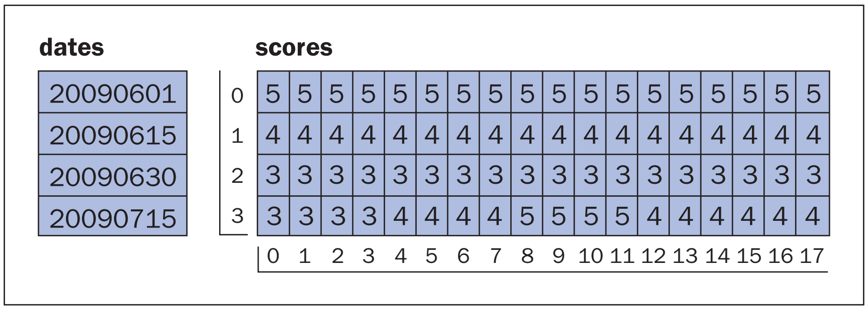
TABLE 12-6
The methods of the `GolfScoreCard` class

`GolfScoreCard` METHOD	WHAT IT DOES
`GolfScoreCard (String fileName)`	Creates a scorecard whose data come from the file named by `fileName`. Throws an `IOException` if the file does not exist or an input error occurs.
`String toString()`	Returns a string representation of the scorecard.
`String highLowDays()`	Returns a string containing the results for the high and low scores for the rounds.
`String bestWorstHoles()`	Returns a string containing the results for the high and low holes, on the average, for the rounds.

We leave the implementation of the driver program that uses these methods as an exercise for you.

The `GolfScoreCard` class represents a card as two arrays. The first array contains the dates read from the input file. The second array is a two-dimensional array. Each row in this array represents a round of 18 holes for a given date. Each column in this array contains the scores on a given hole. Figure 12-9 depicts the state of these data structures after the input of our example **scores.txt** file.

FIGURE 12-9
The two arrays for the golf scores tracking program



dates
20090601
20090615
20090630
20090715

scores	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
1	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
3	3	3	3	3	4	4	4	4	5	5	5	5	4	4	4	4	4	4

The constructor for the `GolfScoreCard` class opens an input stream on the filename, instantiates the two arrays, and transfers the data from the file into the two arrays. The `toString` method builds a string that labels these data and returns the string. The `highLowDays` method also builds and returns a string. This method must search the two-dimensional array of scores for the rows with the lowest and highest sums. These results are returned in a string. The method relies on a private helper method `dayTotal`, which returns the sum of the scores in a given row. We leave the implementation of the `bestWorstHoles` method as an exercise for you. Here is the code for the partially completed `GolfScoreCard` class:

```java
// Example 12.4

import java.util.Scanner;
import java.io.*;

public class GolfScoreCard{

   private int[] dates;
   private int[][] scores;

   // Instantiate and read golf scores from a text file

   public GolfScoreCard(String fileName) throws IOException {

      Scanner fileReader = new Scanner(new File(fileName));

      // Read the number of days

      int numDays = fileReader.nextInt();

      // Instantiate dates and dailyScores
      dates = new int[numDays];
      scores = new int[numDays][18];

      // Read the scores for each day
      //   Date (yyyymmdd), followed by 18 scores
      for (int i = 0; i < numDays; i++){
         dates[i] = fileReader.nextInt() ;
         for (int j = 0; j < 18; j++)
            scores[i][j] = fileReader.nextInt();
      }
      // Close the file
      fileReader.close();
   }

   // Return a string with one line per day.
   // Each line consists of a date and 18 scores
   public String toString(){
      String str = "";
      for (int i = 0; i < dates.length; i++){
         str += "Date: " + dates[i] + " Scores:";
            for (int j = 0; j < 18; j++)
               str += " " + scores[i][j];
         str += "\n";
      }
      return str;
   }

   // Return a string with two lines.
   // The first line contains the date and scores for the best day
   // The second line contains the date and scores for the worst day
   public String highLowDays(){
      // Assume that the first day is the best and worst
      int indexLow = 0;
      int indexHigh = 0;
      int lowTotal = dayTotal(0);
      int highTotal = dayTotal(0);

      // Now consider the remaining days
      for (int i = 1; i < dates.length; i++){
         int todayTotal = dayTotal(i);
         if (todayTotal < lowTotal){
            indexLow = i;
            lowTotal = todayTotal;
         }else if (todayTotal > highTotal){
            indexHigh = i;
            highTotal = todayTotal;
         }
      }

      // Format the return string
      String str = "";
      str += "The best day:  " + dates[indexLow] +
             " score: " + lowTotal + "\n";
      str += "The worst day: " + dates[indexHigh] +
             " score: " + highTotal + "\n";
      return str;
   }
```

```
      // Return the total for the indicated day
      private int dayTotal (int i){
         int total = 0;
         for (int j = 0 ; j < 18; j++)
            total += scores[i][j];
         return total;
      }

      // Other methods go here
   }
```

EXERCISE 12.5

1. Describe a strategy that the computer could use find an empty cell in which to place its letter in tic-tac-toe.

2. Describe a strategy that a computer could use to try to win the game of tic-tac-toe.

3. Describe a strategy for finding the holes with the highest and lowest average scores in the golf scores tracking program.

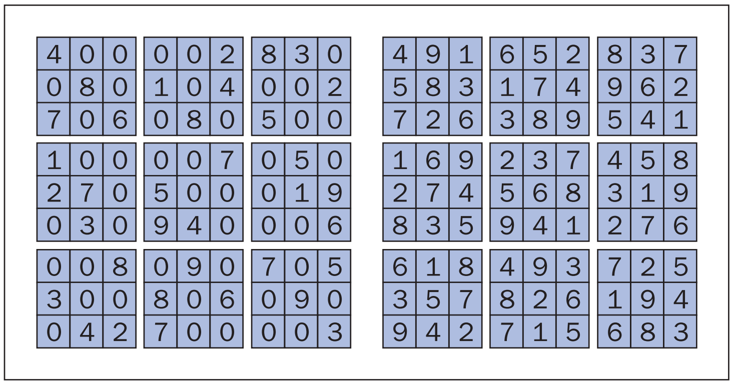
12.6 Case Study: Solving Sudoku Puzzles

Sudoku is a popular single-player puzzle. Sudoku uses a square grid of cells, which is subdivided into nine 3 by 3 subgrids of cells. Initially, some numbers appear in cells, where they must remain as the puzzle is solved. To solve the puzzle, the player fills the remaining cells with the numbers 1–9, such that

- Each 3 by 3 subgrid contains one instance of each number
- Each row and column in the outer 9 by 9 grid contains one instance of each number

Figure 12-10 shows a Sudoku puzzle and its solution. This particular puzzle is considered an easy one to solve. The simplest solution strategy for a puzzle such as this one is to fill in all of the blank cells that admit of a single possibility. This process is then repeated until all of the cells are filled or no more such cells are available.

FIGURE 12-10
A sample Sudoku puzzle and its solution



4	0	0	0	0	2	8	3	0
0	8	0	1	0	4	0	0	2
7	0	6	0	8	0	5	0	0
1	0	0	0	0	7	0	5	0
2	7	0	5	0	0	0	1	9
0	3	0	9	4	0	0	0	6
0	0	8	0	9	0	7	0	5
3	0	0	8	0	6	0	9	0
0	4	2	7	0	0	0	0	3

4	9	1	6	5	2	8	3	7
5	8	3	1	7	4	9	6	2
7	2	6	3	8	9	5	4	1
1	6	9	2	3	7	4	5	8
2	7	4	5	6	8	3	1	9
8	3	5	9	4	1	2	7	6
6	1	8	4	9	3	7	2	5
3	5	7	8	2	6	1	9	4
9	4	2	7	1	5	6	8	3

If the first solution strategy leaves some cells empty, then other strategies can be tried to complete the solution. One such additional strategy is to consider each row. Then, consider each number missing from the row. If the number can only go in one place in the row, then put it there. Repeat this process until there are no further changes.

More difficult puzzles require more complex solution strategies and auxiliary data structures that are beyond the scope of this book. In this case study, we develop a way of representing a Sudoku board and a simple solution strategy, and leave some other strategies as exercises for you.

Request

Write a program that solves easy Sudoku puzzles.

Analysis

The representation of a Sudoku puzzle is stored in a text file. In this file, a 0 indicates a blank cell. The numbers are formatted in nine rows and nine columns and are separated by a single blank space. An extra space and an extra blank line indicate the boundaries of subgrids within the file, as shown in Figure 12-11.

FIGURE 12-11
The format of a Sudoku puzzle in a text file

```
4 0 0  0 0 2  8 3 0
0 8 0  1 0 4  0 0 2
7 0 6  0 8 0  5 0 0

1 0 0  0 0 7  0 5 0
2 7 0  5 0 0  0 1 9
0 3 0  9 4 0  0 0 6

0 0 8  0 9 0  7 0 5
3 0 0  8 0 6  0 9 0
0 4 2  7 0 0  0 0 3
```

At start-up, the program prompts the user for a filename or the letter "q" to quit. The program then attempts to solve the given puzzle, printing the solution or partial solution and indicating that it was solved or not. Figure 12-12 shows a session with our example puzzle. The program continues prompting for filenames and solving puzzles until the user enters "q."

FIGURE 12-12
A session with the Sudoku puzzle solver

```
Initial configuration
4     |     2 | 8 3
  8   | 1   4 |     2
7   6 |   8   | 5
------+-------+------
1     |     7 |   5
2 7   | 5     |   1 9
  3   | 9 4   |     6
------+-------+------
    8 |   9   | 7   5
3     | 8   6 |   9
  4 2 | 7     |     3
Rule 1: results after filling in cells that allow one possibility
4 9 1 | 6 5 2 | 8 3 7
5 8 3 | 1 7 4 | 9 6 2
7 2 6 | 3 8 9 | 5 4 1
------+-------+------
1 6 9 | 2 3 7 | 4 5 8
2 7 4 | 5 6 8 | 3 1 9
8 3 5 | 9 4 1 | 2 7 6
------+-------+------
6 1 8 | 4 9 3 | 7 2 5
3 5 7 | 8 2 6 | 1 9 4
9 4 2 | 7 1 5 | 6 8 3
Enter the name of a puzzle file or Q to quit:
```

Classes

The application consists of a data model class named `Sudoku` and a view class named `PlaySudoku`. The `PlaySudoku` class handles interactions with the user. The `Sudoku` class represents the puzzle's board and implements the strategies for solving puzzles. Table 12-7 lists the `public` methods for the `Sudoku` class.

TABLE 12-7
The interface for the `Sudoku` class

`Sudoku` METHOD	WHAT IT DOES
`Sudoku(String fileName)`	Creates a Sudoku puzzle whose numbers come from the file named by `fileName`. Throws an `IOException` if the file does not exist or an input error occurs.
`String toString()`	Returns a string representation of the puzzle.
`void rule1()`	Attempts to solve the puzzle using a simple solution strategy.
`int countNumberOfZeros()`	Returns the number of 0s in the puzzle.

Note that there is a single strategy for solving a puzzle, implemented as a method named `rule1`. The programmer can develop other methods to solve a puzzle by adding them to the `public` methods of the `Sudoku` class.

Design

The `PlaySudoku` class implements a simple driver loop for inputting and solving puzzles. Like the driver of golf scores program discussed earlier, this driver repeatedly takes an input filename from the user. The driver then instantiates a Sudoku object which reads a puzzle from the named file. The driver then applies rule methods to the Sudoku object until the puzzle is solved or until there are no further improvements.

The `Sudoku` class includes a fairly simple constructor that instantiates and fills a two-dimensional array with numbers read from a text file. The class's `toString` method builds a string like the ones shown in Figure 12-12. Most of the design work involves developing pseudocode for rule 1. Because this rule is fairly complex, we decompose it into a top-level task and several subtasks. At the top level, the `rule1` method repeatedly traverses the array, checking locations that contain a zero to determine if they can be filled by exactly one value. Here is the pseudocode:

```
Method rule1
   Set done to false
   While not done
      Set done to true
      For each row
         For each column
            If array[row][column] equals 0
               Set value to the unique value for (row, column)
               If the value does not equal 0
                  Set array[row][column] to the value
                  Set done to false
```

This process leaves the finding of a unique value for a given row and column to another method named `findUniqueValueFor`. This method iterates through the numbers 1–9 to find one that does not cause a conflict at a given row and column:

```
Method findUniqueValueFor(row, column)
   Set value to 0
   For each v from 1 to 9
      If v does not cause a conflict at (row, column)
         If value equals 0
            Set value to v
         Else
            Return 0 because there is more than
                     one possible value
   Return value
```

Finally, the method `probeIsOkay` determines whether or not a given value causes a conflict at a given row and column:

```
Method probeIsOkay(probe, row, column)
   If there is a conflict in the row
      Return false
   If there is a conflict within the column
```

```
      Return false
   If there is a conflict within the subgrid
   that contains (row, column)
      Return false
   Return true
```

Implementation of `PlaySudoku`

The `PlaySudoku` class implements the user interface for the game. The main method repeatedly asks the user for a filename, creates a `Sudoku` object from that file, and calls a solve method to solve the puzzle. The helper method `solve` is structured to allow the successive application of Sudoku methods to try to reduce the number of zeros on the board. In this version of the program, only the `rule1` method is applied.

```java
import java.util.Scanner;
import java.io.*;

   public class PlaySudoku {

      public static void main(String[] args) throws IOException{
         while (true){
            Scanner reader = new Scanner(System.in);
            System.out.print("Enter the name of a puzzle " +
                             "file or Q to quit: ");
            String fileName = reader.next();
            if (fileName.equalsIgnoreCase("q"))
               break;
            Sudoku puzzle = new Sudoku(fileName);
            solve(puzzle);
         }
      }

      private static void solve(Sudoku puzzle){
         System.out.println("Initial configuration");
         System.out.print(puzzle);
         int numZeros = puzzle.countNumberOfZeros();
         int oldNumZeros = numZeros + 1;

         // Apply the rules until there is no further improvement
         while(numZeros < oldNumZeros){
            oldNumZeros = numZeros;
            puzzle.rule1();
            System.out.println("Rule 1: results after filling" +
                           " in cells that allow one possibility");
            System.out.print(puzzle);
            if (puzzle.countNumberOfZeros() == 0)
               return;

            // Attempt to solve by applying another rule here

            numZeros = puzzle.countNumberOfZeros();
         }
      }
}
```

Implementation of Sudoku

The methods in the Sudoku class implement the strategies outlined in the design phase. Here is the code:

```
import java.util.Scanner;
import java.io.*;

public class Sudoku {

   private int[][] matrix;
   // The puzzle is stored in a 9x9 matrix.
   // Zeros indicate cells that don't yet have a value

   // Instantiate and read a puzzle from a text file
   public Sudoku(String fileName) throws IOException {
      Scanner fileReader = new Scanner(new File(fileName));
      // Read the puzzle
      matrix = new int[9][9];
        for (int i = 0; i < 9; i++)       // 9 rows
           for (int j = 0; j < 9; j++)    // 9 columns
                 matrix[i][j] = fileReader.nextInt();
      fileReader.close();
   }

   // Return a string representation of the puzzle.
   // Use spaces rather than zeros.
   public String toString(){
      String str = "\n";
      for (int i = 0; i < 9; i++){
         for (int j = 0; j < 9; j++){
            if (matrix[i][j] != 0)
               str += matrix[i][j] + " ";
            else
               str += "  ";
            if (j == 2 || j == 5)
               str += "| ";
         }
         str += "\n";
         if (i == 2 || i == 5)
            str += "---------------------\n";
      }
      str += "\n";
      return str;
   }

   //   Fill in all blank entries that admit of a single possibility.
   //   Repeat until there are no further changes
   public void rule1(){
      // Apply this rule until there is a complete
      // pass through the matrix without changing any
      // of the cells.
       boolean done = false;
```

```
    while (!done){
       done = true;
       // Traverse the puzzle row by row
       // and column by column
       for (int i = 0; i < 9; i++){
          for (int j = 0; j < 9; j++){
          // If a cell hasn't been resolved
          // (contains zero), then set it to the
          // unique value that goes in the cell. If
          // there is more than one value possible,
          // don't change the cell.
             if (matrix[i][j] == 0){
                int value = findUniqueValueFor(i, j);
                if (value != 0){
                    matrix[i][j] = value;
                    done = false;
                }
             }
          }
       }
    }
}

// Return the unique value that satisfies cell
// [i][j] or 0 if there is none.
private  int findUniqueValueFor(int i, int j){
   int value = 0;
   // Try each of the candidate values
   for (int v = 1; v <= 9; v++){
      if (probeIsOkay(v, i, j)){
      // If the value doesn't cause a conflict
         // If this is the first value that works,
         if (value == 0)
            value = v;          // use it.
         // Else there is more than one possible value
         else
            return 0;
      }
   }
   return value;       // Return the unique value
}

// Determine if a particular value can be placed in
// a particular cell
private  boolean probeIsOkay(int probe, int i, int j){
   // Check row i for conflicts
   for (int jj = 0; jj < 9; jj++)
      if (jj != j){
         if (probe == matrix[i][jj])
            return false;
      }
```

```
            // Check column j for conflicts
            for (int ii = 0; ii < 9; ii++)
               if (ii != i){
                  if (probe == matrix[ii][j])
                     return false;
               }

            // Check box containing [i][j] for conflicts
           int topLeftRow = i - i%3;
           int topLeftColumn = j - j%3;
           for (int ii = topLeftRow; ii < topLeftRow + 3; ii++)
              for (int jj = topLeftColumn; jj < topLeftColumn + 3; jj++)
                 if (ii != i || jj != j){
                    if (probe == matrix[ii][jj])
                       return false;
                   }

            // There are no conflicts
            return true;
        }
    }
```

12.7 Graphics and GUIs: Menus

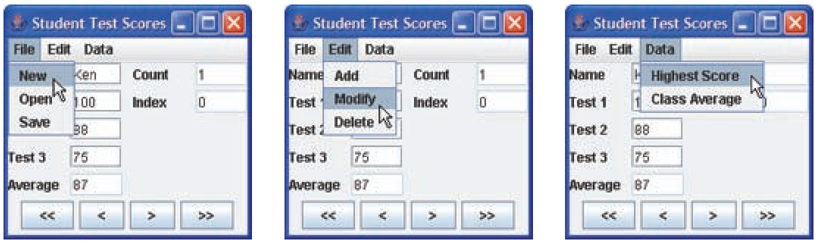
Extra Challenge

This Graphics and GUIs section gives you the opportunity to explore concepts and programming techniques required to develop modern graphics applications and graphical user interfaces. This material is not required in order to proceed with the other chapters of the book.

In this section, we add drop-down menus to our growing repertoire of GUI features. As the number of commands grows, an application's user interface can become increasingly cluttered with a confusing jumble of command buttons. For example, the student test scores application in Section 10.10 already has eight buttons, but many more commands would be typical in a realistic application. Drop-down menus help to economize on a window's real estate by hiding commands under visible labels that expand only when needed.

A drop-down menu system consists of a menu bar, a number of menus, and for each menu, several selections. It is also possible to have submenus, but we ignore these for now. It is easy to add a drop-down menu system to an application. For instance, the student test scores program currently has two rows of buttons, one for data access and another for navigation. We would like to add commands for deleting a student and for transferring the database of students to and from a file. The new commands and the data access commands can be placed under three drop-down menus named File, Edit, and Data. The selections under File are the standard New, Open, and Save. The selections under Edit are Add, Modify, and Delete. The selections under Data are Highest Score and Class Average. The new user interface is shown in Figure 12-13.

FIGURE 12-13
The new user interface for the student test scores program



We create a menu item object for each menu selection (class `JMenuItem`), a menu object for each menu (class `JMenu`), and the menu bar object in which all of the menu objects will appear (class `JMenuBar`). We then add all of the objects to their appropriate containers. The menus and selections are displayed in the order in which they are added in the code. The following code shows a truncated version of the `TestScoresView` class from Section 10.10, which adds the three menus to the application's window:

```
// Example 12.5: TestScoresView class (with menus)

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class TestScoresView extends JFrame{

   // >>>>>>> The model <<<<<<<<

   // Declare the model
   private TestScoresModel model;

   // >>>>>>> The view <<<<<<<<

   // Declare and instantiate the menu items
   private JMenuItem newMI        = new JMenuItem("New");
   private JMenuItem openMI       = new JMenuItem("Open");
   private JMenuItem saveMI       = new JMenuItem("Save");
   private JMenuItem addMI        = new JMenuItem("Add");
   private JMenuItem modifyMI     = new JMenuItem("Modify");
   private JMenuItem deleteMI     = new JMenuItem("Delete");
   private JMenuItem highScoreMI  = new JMenuItem("Highest Score");
   private JMenuItem aveScoreMI   = new JMenuItem("Class Average");
   // Code for creating the other window objects

   // Constructor
   public TestScoresView(TestScoresModel m){
      model = m;
      // Organize and install the menu system
      JMenu fileMenu =  new JMenu("File");
      fileMenu.add(newMI);
      fileMenu.add(openMI);
      fileMenu.add(saveMI);
```

```
         JMenu editMenu =  new JMenu("Edit");
         editMenu.add(addMI);
         editMenu.add(modifyMI);
         editMenu.add(deleteMI);
         JMenu dataMenu =  new JMenu("Data");
         dataMenu.add(highScoreMI);
         dataMenu.add(aveScoreMI);
         JMenuBar bar = new JMenuBar();
         bar.add(fileMenu);
         bar.add(editMenu);
         bar.add(dataMenu);
         setJMenuBar(bar);
         // Code for installing the other window objects
         // Note: The north panel of buttons has been deleted
         // Attach listeners to buttons and menu items
         addMI.addActionListener(new AddListener());
         previousButton.addActionListener(new PreviousListener());
         // Set window attributes
         setTitle("Student Test Scores");
         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
         pack();
         setVisible(true);
      }
      // Code for the rest of the class
   }
```

Like buttons, menu items emit action events when selected, so the programmer simply attaches action listeners for the appropriate tasks to the menu items. Therefore, the `listener` classes in the existing code need no modification whatsoever, although new listeners must be defined for the file operations. The completion of the new version is left as an exercise.

SUMMARY

In this chapter, you learned:

- A linear search is a simple search method that works well for small- and medium-sized arrays.
- A binary search is a clever search method that works well for large arrays but assumes that the elements are sorted.
- Comparisons of objects are accomplished by implementing the `Comparable` interface, which requires the `compareTo` method.
- Selection sort, bubble sort, and insertion sort are simple sort methods that work well for small- and medium-sized arrays.
- Insertions and removals of elements at arbitrary positions are complex operations that require careful design and implementation.
- Two-dimensional arrays store values in a row-and-column arrangement similar to a table.

VOCABULARY*Review*

Define the following terms:

Binary search	Linear search	Ragged array
Bubble sort	Multidimensional array	Selection sort
Insertion sort	One-dimensional array	Two-dimensional array

REVIEW*Questions*

MULTIPLE CHOICE QUESTIONS

Select the correct answer from the list of possibilities.

1. The faster search of an array of elements for a target element is the __________.
   a. linear search
   b. binary search
   c. selection sort

2. Which sort algorithm includes a search for the largest or smallest element?
   a. bubble sort
   b. insertion sort
   c. selection sort

3. The insertion of an element into an array requires __________.
   a. opening a hole in the array by shifting elements to the right of the target position
   b. closing a hole in the array by shifting elements to the right of the target position

4. The number of elements currently available and accessible in an array is called its __________.
   a. logical size
   b. physical size

5. A row-major traversal of a two-dimensional grid visits all of the positions in a __________.
   a. row before moving to the next row
   b. column before moving to the next column

PROJECTS

In first three projects that follow, you are asked to implement a new class called `ArrayList`. This collection class is like an array, but tracks its own logical size and supports insertions and removals of objects. A more complete version of this class is already defined in the package `java.util`, which we explore in Chapter 14.

PROJECT 12-1

Define a class named `ArrayList`. This class represents a list of objects and supports random access to its objects via a numeric index position. Thus, it should contain an array of `Object`. For this exercise, include an instance variable for the array, an instance variable to track the number of objects currently accessible in the array (its logical size), three constructors, and a `toString` method. The default constructor should create an array of five objects, all initially `null`. The second constructor expects an integer parameter. This constructor uses this integer as the initial length of the array. The third constructor expects an array of objects as an argument. It copies these objects to a new array instance variable of the same length. The count (logical size) of the accessible objects should be initialized appropriately in each constructor. The `toString` method builds and returns a string containing the string representations of the objects currently accessible in the array. When the logical size is 0, this is the empty string.

PROJECT 12-2

Add the methods `size` and `get` to the `ArrayList` class of Project 12-1. The `size` method expects no arguments and returns the number of objects currently accessible in the list. The `get` method expects an integer argument and returns the object at that position in the list. If the integer argument is less than 0 or greater than or equal to the size of the list, then the method should throw an `IndexOutOfBoundsException`.

PROJECT 12-3

Add the methods `add` and `remove` to the `ArrayList` class of Project 12-2. The `remove` method expects an integer argument and removes and returns the object at that position in the list. If the integer argument is less than 0 or greater than or equal to the size of the list, then the `remove` method should throw an `IndexOutOfBoundsException`. The `add` method expects an integer and an object as arguments and inserts that object at that position in the list, if there is room for the new object in the underlying array. If the integer argument to `add` is less than or equal to 0, then the object goes at the beginning of the list. If the integer argument is greater than or equal to the size of the list, then the object goes at the end of the list. If there is room for the new object in the underlying array, the `add` method returns `true`; otherwise, it returns `false`.

PROJECT 12-4

Complete the tic-tac-toe example presented in this chapter by adding a method that tests the board for a winner. This method should build a three-character string from each row, column, and diagonal in the board and then search this set of strings for a match to the string “XXX” or “OOO”. You should feel free to decompose this task into subtasks, such as building a string from a row, building a string from a column, and searching an array of strings for a given string, using `private` helper methods where appropriate.

PROJECT 12-5

Modify the tic-tac-toe game of Project 12-4 so that the computer automatically makes the moves of the second player. The computer uses a strategy of marking an empty square chosen at random.

PROJECT 12-6

Modify the tic-tac-toe game of Project 12-5 so that the computer tries to win. That is, the computer should make an attempt to block the opponent from getting three letters in a row and also attempt to locate a position that will give it three letters in a row.

PROJECT 12-7

Complete the methods for the golf scores tracking program presented in this chapter.

PROJECT 12-8

A par in golf is the number of shots normally expected on a given hole. Par for the course is the sum of the shots expected for all of the holes in a round. A handicap is the number of shots that a player averages over par for several rounds. The minimum handicap is 0. Add a method `getHandicap` to the `GolfScoreCard` class. This method expects an integer parameter that represents par for the course. The method returns the handicap for that scorecard.

PROJECT 12-9

Add a new method named `rule2` to the `Sudoku` class of the case study in this chapter. This method implements the following strategy:

- Consider each row.
- Consider each number missing from the row.
- If the number can go in only one place within the row, put it there.
- Repeat until there are no further changes.

You should run the new method if the `rule1` method fails to eliminate all the zeros from the board.

PROJECT 12-10

Add a new method named `rule3` to the `Sudoku` class of the case study in this chapter. This method implements the following strategy:

- Consider each column.
- Consider each number missing from the column.
- If the number can go in only one place within the column, put it there.
- Repeat until there are no further changes

You should run the new method if the `rule1` method and `rule2` method fail to eliminate all the zeros from the board.

PROJECT 12-11

A magic square is a two-dimensional array of positive integers such that the sum of each row, column, and diagonal is the same constant. The following example is a magic square whose constant is 34:

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

Write a program that takes 16 integers as inputs. The program should determine whether or not the square is a magic square and display the result.

PROJECT 12-12

Pascal's triangle can be used to recognize coefficients of a quantity raised to a power. The rules for forming this triangle of integers are such that each row must start and end with a 1, and each entry in a row is the sum of the two values diagonally above the new entry. Thus, four rows of Pascal's triangle are

```
            1
        1       1
    1       2       1
1       3       3       1
```

This triangle can be used as a convenient way to get the coefficients of a quantity of two terms raised to a power (binomial coefficients). For example

$(a+b)^3=1\times a^3+3a^2b+3ab^2+1\times b^3$

where the coefficients 1, 3, 3, and 1 come from the fourth row of Pascal's triangle.

Write a program that takes the number of rows (up to, say, 10) as input and displays Pascal's triangle for those rows.

PROJECT 12-13

In the game of Penny Pitch, a two-dimensional board of numbers is laid out as follows:

```
1  1  1  1  1

1  2  2  2  1

1  2  3  2  1

1  2  2  2  1

1  1  1  1  1
```

A player tosses five pennies on the board, aiming for the number with the highest value. At the end of the game, the sum total of the tosses is returned. Develop a program that plays this game. The program should display the board and then perform the following steps each time the user presses Enter:

- Generate two random numbers for the row and column of the toss.
- Add the number at this position to a running total.
- Display the board, replacing the numbers with Ps where the pennies land.

(*Hint*: You should use a two-dimensional array of `Square` objects for this problem. Each square contains a number like those shown and a Boolean flag that indicates whether or not a penny has landed on that square.)

PROJECT 12-14

Complete the GUI-based student test scores program developed in Section 12-8.

PROJECT 12-15

Modify the user interface for the tic-tac-toe program for two human players so that it uses a GUI. As before, the program maintains the data model of the game as a `TTTBoard` object. But now the game board should be displayed in a panel that contains a two-dimensional grid of buttons. The labels of the buttons initially are blanks. The players take turns pressing these buttons, and the program tracks whose turn it is, as before. When a player presses a blank button, its label changes to that player's letter and the `TTTBoard` object is updated. Presses of non-blank buttons are ignored. When a win or tie occurs, the players are notified with a message box. You should also include a separate New Game button that resets the display to blank buttons and resets the data model for the game. (*Hints*: Place the board buttons in a two-dimensional array and use this array to reset the buttons when the user wants a new game. Define an action listener class that contains references to a board button and its row and column in the grid. Attach a new instance of this class to each board button. Use the methods `getText` and `setText` to observe and modify a button's label.)

CRITICAL *Thinking*

An application needs to load numbers from a text file into an array. The text file contains at least 100 numbers and at most 1000 numbers. Suggest a method for loading these numbers into the array.

ARRAYS AND CLASSES

REVIEW *Questions*

TRUE/FALSE

Circle T if the statement is true or F if it is false.

T F 1. In Java, an array is an ordered collection of methods.

T F 2. Because arrays are objects, two variables can refer to the same array.

T F 3. The length of a Java array is fixed at compile time.

T F 4. One responsibility of the view portion of a program is to instantiate and arrange window objects.

T F 5. Static variables are associated with a class, not with its instances.

T F 6. Subclasses can see both public and private names declared in their parent classes.

T F 7. If you need to use the cast operator, remember that you never cast down, only cast up.

FILL IN THE BLANK

Complete the following sentences by writing the correct word or words in the blanks provided.

1. A(n) __________ is a collection of similar items or elements that are ordered by position.
2. An item's position within an array is called its __________ or __________.
3. Two arrays in which the corresponding elements are related are called __________ arrays.
4. The number of elements currently stored and used within an array is called its __________ size.
5. Classes that are never instantiated are called __________.
6. Java organizes classes in a(n) __________.
7. The `toString` message, which is understood by every object, no matter which class it belongs to, is a good example of __________.

WRITTEN QUESTIONS

Write a brief answer to each of the following questions.

1. Write statements for the following items that declare array variables and assign the appropriate array objects to them.
   A. `intNumbers`, an array of 5 integers

   B. `realNumbers`, an array of 100 real numbers

   C. `bools`, an array of 10 Booleans

   D. `words`, an array of 20 strings

2. Write a `for` loop that initializes an array of 10 integers to the first 10 positive integers.

3. Repeat question 2, but use an initializer list.

4. There are several ways in which methods in a subclass can be related to methods in its superclass. Describe at least two of these.

5. Explain why one would use `static` variables and `static` methods.

PROJECTS

PROJECT 1

Write a program that takes 10 floating-point numbers as inputs. The program then displays the average of the numbers followed by all of the numbers that are greater than the average. As part of your design, write a method that takes an array of doubles as a parameter and returns the average of the data in the array.

PROJECT 2

Write a program to keep statistics for a basketball team consisting of 10 players. Statistics for each player should include shots attempted, shots made, and shooting percentage; free throws attempted, free throws made, and free throw percentage; offensive rebounds and defensive rebounds; assists; turnovers; and total points. Place these data in parallel arrays. Appropriate team totals should be listed as part of the output.

PROJECT 3

Modify the program of Project 2 so that it uses a two-dimensional array instead of the parallel arrays for the statistics.

PROJECT 4

A summation method returns the sum of the numbers from a lower bound to an upper bound. Write a static recursive method for summations and run it with a `tester` program that displays the values of the parameters on each call and the returned value on each call.

CRITICAL *Thinking*

A bank provides several kinds of accounts, among them checking accounts and saving accounts. Design a simple banking system data model that represents these accounts. Be sure to make use of abstract classes, inheritance, polymorphism, and encapsulation. The result of your work should be a set of class summary boxes for the data model of the banking system.