7.15 Class `Arrays`

Class `Arrays` helps you avoid reinventing the wheel by providing `static` methods for common array manipulations. These methods include `sort` for *sorting* an array (i.e., arranging elements into ascending order), `binarySearch` for *searching* a *sorted* array (i.e., determining whether an array contains a specific value and, if so, where the value is located), `equals` for *comparing* arrays and `fill` for *placing values into an array*. These methods are overloaded for primitive-type arrays and for arrays of objects. Our focus in this section is on using the built-in capabilities provided by the Java API. Chapter 19, Searching, Sorting and Big O, shows how to implement your own sorting and searching algorithms, a subject of great interest to computer-science researchers and students, and to developers of high-performance systems.

Figure 7.22 uses `Arrays` methods `sort`, `binarySearch`, `equals` and `fill`, and shows how to *copy* arrays with class `System`'s `static` `arraycopy` **method**. In `main`, line 9 sorts the elements of array `doubleArray`. The `static` method `sort` of class `Arrays` orders the array's elements in *ascending* order by default. We discuss how to sort in *descending* order later in the chapter. Overloaded versions of `sort` allow you to sort a specific range of elements within the array. Lines 10–14 output the sorted array.

```
1  // Fig. 7.22: ArrayManipulations.java
2  // Arrays class methods and System.arraycopy.
3  import java.util.Arrays;
4
5  public class ArrayManipulations {
6     public static void main(String[] args) {
7        // sort doubleArray into ascending order
8        double[] doubleArray = {8.4, 9.3, 0.2, 7.9
9        Arrays.sort(doubleArray);
10       System.out.printf("%ndoubleArray: ");
11
12       for (double value : doubleArray) {
13          System.out.printf("%.1f ", value);
14       }
15
16       // fill 10-element array with 7s
17       int[] filledIntArray = new int[10];
18       Arrays.fill(filledIntArray, 7);
19       displayArray(filledIntArray, "filledIntArr
20
21       // copy array intArray into array intArray
22       int[] intArray = {1, 2, 3, 4, 5, 6};
23       int[] intArrayCopy = new int[intArray.leng
24       System.arraycopy(intArray, 0, intArrayCopy
25       displayArray(intArray, "intArray");
26       displayArray(intArrayCopy, "intArrayCopy")
27
28       // compare intArray and intArrayCopy for e
29       boolean b = Arrays.equals(intArray, intArr
30       System.out.printf("%n%nintArray %s intArra
31          (b ? "==" : "!="));
32
33       // compare intArray and filledIntArray for
34       b = Arrays.equals(intArray, filledIntArray
35       System.out.printf("intArray %s filledIntAr
36          (b ? "==" : "!="));
37
38       // search intArray for the value 5
39       int location = Arrays.binarySearch(intArra
40
```

```
41         if (location <= 0) {
42            System.out.printf(
43               "Found 5 at element %d in intArray%n
44         }
45         else {
46            System.out.println("5 not found in intA
47         }
48
49       // search intArray for the value 8763
50       location = Arrays.binarySearch(intArray, 8
51
52         if (location >= 0) {
53            System.out.printf(
54               "Found 8763 at element %d in intArra
55         }
56         else {
57            System.out.println("8763 not found in i
58         }
59      }
60
61      // output values in each array
62      public static void displayArray(int[] array,
63         System.out.printf("%n%s: ", description);
64
65         for (int value : array) {
66            System.out.printf("%d ", value);
67         }
68      }
69   }
```

```
doubleArray: 0.2 3.4 7.9 8.4 9.3
filledIntArray: 7 7 7 7 7 7 7 7 7 7
intArray: 1 2 3 4 5 6
intArrayCopy: 1 2 3 4 5 6

intArray == intArrayCopy
intArray != filledIntArray
Found 5 at element 4 in intArray
```

```
8763 not found in intArray
```

Fig. 7.22

`Arrays` class methods and `System.arraycopy`.

Line 18 calls `static` method `fill` of class `Arrays` to populate all 10 elements of `filledIntArray` with 7s. Overloaded versions of `fill` allow you to populate a specific range of elements with the same value. Line 19 calls our class's `displayArray` method (declared at lines 62–68) to output the contents of `filledIntArray`.

Line 24 copies the elements of `intArray` into `intArrayCopy`. The first argument (`intArray`) passed to `System` method `arraycopy` is the array from which elements are to be copied. The second argument (`0`) is the index that specifies the *starting point* in the range of elements to copy from the array. This value can be any valid array index. The third argument (`intArrayCopy`) specifies the *destination array* that will store the copy. The fourth argument (`0`) specifies the index in the destination array *where the first copied element should be stored*. The last argument specifies the *number of elements to copy* from the array in the first argument. In this case, we copy all the elements in the array.

Lines 29 and 34 call `static` method `equals` of class `Arrays` to determine whether all the elements of two arrays

are equivalent. If the arrays contain the same elements in the same order, the method returns `true`; otherwise, it returns `false`.

Error-Prevention Tip 7.3

When comparing array contents, always use `Arrays.equals(array1, array2)`, *which compares the two arrays' contents, rather than* `array1.equals(array2)`, *which compares whether* `array1` *and* `array2` *refer to the same array object.*

Lines 39 and 50 call `static` method `binarySearch` of class `Arrays` to perform a binary search on `intArray`, using the second argument (`5` and `8763`, respectively) as the key. If the value is found, `binarySearch` returns the index of the element; otherwise, `binarySearch` returns a negative value. The negative value returned is based on the search key's *insertion point*—the index where the key would be inserted in the array if we were performing an insert operation. After `binarySearch` determines the insertion point, it changes its sign to negative and subtracts `1` to obtain the return value. For example, in Fig. 7.22, the insertion point for the value `8763` is the element with index `6` in the array. Method `binarySearch` changes the insertion point to `-6`, subtracts `1` from it and returns the value `-7`. Subtracting `1` from the insertion point guarantees that method `binarySearch` returns positive values (>= 0) if and only if the key is found.

This return value is useful for inserting elements in a sorted array. Chapter 19 discusses binary searching in detail.

 Common Programming Error 7.6

Passing an unsorted array to `binarySearch` *is a logic error—the value returned is undefined.*

Java SE 8—Class `Arrays` Method `parallelSort`

8

The `Arrays` class now has several new "parallel" methods that take advantage of multi-core hardware. `Arrays` method `parallelSort` can sort large arrays more efficiently on multi-core systems. In Section 23.12, we create a very large array and use features of the Date/ Time API to compare how long it takes to sort the array with `sort` and `parallelSort`.