7.8 Passing Arrays to Methods

This section demonstrates how to pass arrays and individual array elements as arguments to methods. To pass an array argument to a method, specify the name of the array *without any brackets*. For example, if array `hourlyTemperatures` is declared as

```
double[] hourlyTemperatures = new double[24];
```

then the method call

```
modifyArray(hourlyTemperatures);
```

passes the reference of array `hourlyTemperatures` to method `modifyArray`. Every array object “knows” its own length. Thus, when we pass an array object’s reference into a method, we need not pass the array length as an additional argument.

For a method to receive an array reference through a method call, the method’s parameter list must specify an *array parameter*. For example, the method header for method `modifyArray` might be written as

```
void modifyArray(double[] b)
```

indicating that `modifyArray` receives the reference of a `double` array in parameter `b`. The method call passes array `hourlyTemperature`'s reference, so when the called method uses the array variable `b`, it *refers to* the same array object as `hourlyTemperatures` in the caller.

When an argument to a method is an entire array or an individual array element of a reference type, the called method receives a *copy* of the reference. However, when an argument to a method is an individual array element of a primitive type, the called method receives a copy of the element's *value*. Such primitive values are called **scalars** or **scalar quantities**. To pass an individual array element to a method, use the indexed name of the array element as an argument in the method call.

Figure 7.13 demonstrates the difference between passing an entire array and passing a primitive-type array element to a method. Method `main` invokes `static` methods `modifyArray` (line 18) and `modifyElement` (line 30). Recall that a `static` method can invoke other `static` methods of the same class without using the class name and a dot (`.`).

```
1   // Fig. 7.13: PassArray.java
2   // Passing arrays and individual array elements
3
4   public class PassArray {
5      // main creates array and calls modifyArray a
```

```
    public static void main(String[] args) {
        int[] array = {1, 2, 3, 4, 5};

        System.out.printf(
           "Effects of passing reference to entire
           "The values of the original array are:%

        // output original array elements
           for (int value : array) {
            System.out.printf(" %d", value);
           }

        modifyArray(array); // pass array referenc
        System.out.printf("%n%nThe values of the m

        // output modified array elements
           for (int value : array) {
            System.out.printf(" %d", value);
           }

        System.out.printf(
           "%n%nEffects of passing array element v
           "array[3] before modifyElement: %d%n",

        modifyElement(array[3]); // attempt to mod
        System.out.printf(
           "array[3] after modifyElement: %d%n", a
     }

    // multiply each element of an array by 2
    public static void modifyArray(int[] array2)
       for (int counter = 0; counter < array2.len
           array2[counter] *= 2;
        }
     }

    // multiply argument by 2
    public static void modifyElement(int element)
        element *= 2;
        System.out.printf(
```

```
46             "Value of element in modifyElement: %d%
                     47      }
                      48   }
```

```
Effects of passing reference to entire array:
    The values of the original array are:
                1   2   3   4   5

    The values of the modified array are:
                2   4   6   8   10

   Effects of passing array element value:
      array[3] before modifyElement: 8
    Value of element in modifyElement: 16
      array[3] after modifyElement: 8
```

Fig. 7.13

Passing arrays and individual array elements to methods.

The enhanced `for` statement in lines 14–16 outputs `array`'s elements. Line 18 invokes `modifyArray` (lines 36–40), passing `array` as an argument. The method receives a copy of `array`'s reference and uses it to multiply each of `array`'s elements by 2. To prove that `array`'s elements were modified, lines 22–24 output the elements again. As the output shows, method `modifyArray` doubled the value of each element. We could *not* use the enhanced `for` statement in lines 37–39 because we're modifying the array's elements.

Figure 7.13 next demonstrates that when a copy of an individual primitive-type array element is passed to a method, modifying the *copy* in the called method does *not* affect the original value of that element in the calling method's array. Lines 26–28 output the value of `array[3]` *before* invoking method `modifyElement`. Remember that the value of this element is now 8 after it was modified in the call to `modifyArray`. Line 30 calls method `modifyElement` and passes `array[3]` as an argument. Remember that `array[3]` is actually one `int` value (8) in `array`. Therefore, the program passes a copy of `array[3]`'s value. Method `modifyElement` (lines 43–47) multiplies the value received as an argument by 2, stores the result in its parameter `element`, then outputs the value of `element` (16). Since method parameters, like local variables, cease to exist when the method in which they're declared completes execution, the method parameter `element` is destroyed when `modifyElement` terminates. When the program returns control to `main`, lines 31–32 output the *unmodified* value of `array[3]` (i.e., 8).