CHAPTER: 06

Arrays

Objectives:

By the end of this chapter you should be able to:

- *create* ***arrays****;*
- *use* **`for`** *loops to process arrays;*
- *use an enhanced* **`for`** *loop to process an array;*
- *use arrays as method inputs and outputs;*
- *develop routines for accessing and manipulating arrays;*

6.1 Introduction

In previous chapters we have shown you how to create variables and store data in them. In each case the variables created could be used to hold a *single* item of data. How, though, would you deal with a situation in which you had to create and handle a very large number of data items?

An obvious approach would be just to declare as many variables as you need. Declaring a large number of variables is a nuisance but simple enough. For example, let's consider a very simple application that records seven temperature readings (one for each day of the week):

```
public class TemperatureReadings
{
   public static void main(String[] args)
   {
      // declare 7 variables to hold readings
      double temperature1, temperature2, temperature3, temperature4, temperature5,
             temperature6, temperature7;
      // more code will go here
   }
}
```

Here we have declared seven variables each of type **`double`** (as temperatures will be recorded as real numbers). So far so good. Now to write some code that allows the user to enter values for these temperatures. Getting one temperature is easy (assuming we have created a `Scanner` object `keyboard`), as shown below:

```
System.out.println("max temperature for day 1 ?");
temperature1 = keyboard.nextDouble();
```

But how would you write the code to get the second temperature, the third temperature and all the remaining temperatures? Well, you could repeat the above pair of lines for each temperature entered, but surely you've got better things to do with your time!

Essentially you want to repeat the same pair of lines seven times. You already know that a **`for`** loop is useful when repeating lines of code a fixed number of times. Maybe you could try using a **`for`** loop here?

```
for (int i=1; i<=7; i++)
{
   // what goes here?
}
```

This looks like a neat solution, but the problem is that there is no obvious instruction we could write in the **`for`** loop that will allow a value to be entered into a *different* variable each time the loop repeats. As things stand there is no way around this, as each variable has a *distinct* name.

Ideally we would like each variable to be given the *same* name (`temperature`, say) so that we could use a loop here, but we would like some way of being able to distinguish between each successive variable. In fact, this is exactly what an **array** allows us to do.

6.2 Creating an array

An array is a special data type in Java that can be thought of as a *container* to store a *collection of items*. These items are sometimes referred to as the **elements** of the array. All the elements stored in a particular array must be of the *same type* but there is no restriction on which type this is. So, for example, an array can be used to hold a collection of **`int`** values or a collection of **`char`** values, but it cannot be used to hold a mixture of **`int`** and **`char`** values.

Let's look at how to use arrays in your programs. First you need to know how to create an array. Array creation is a two-stage process:

1 declare an array variable;

2 allocate memory to store the array elements.

An array variable is declared in much the same way as a simple variable except that a pair of square brackets is added after the type. For example, if an array was to hold a collection of integer variables it could be declared as follows:

```
int[] someArray;
```

Here a name has been given to the array in the same way you would name any variable. The name we have chosen is `someArray`. If the square brackets were missing in the above declaration this would just be a simple variable capable of holding a *single* integer value only. But the square brackets indicate this variable is an array allowing *many* integer values to be stored.

So, to declare an array `temperature` containing **`double`** values, you would write the following:

```
double[] temperature;
```

At the moment this simply defines `temperature` to be a variable that can be *linked* to a collection of **`double`** values. The `temperature` variable itself is said to hold a **reference** to the array elements. A reference is a variable that holds a *location* in the computer's memory (known as a memory *address*) where data is stored, rather than the data itself. This illustrated in figure 6.1 below:

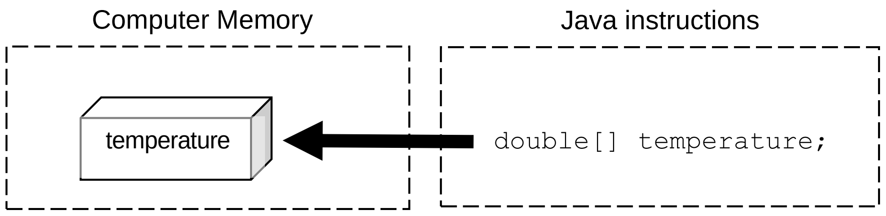


Fig 6.1. The effect on computer memory of declaring an array reference

At the moment the memory address stored in the `temperature` reference is not meaningful as the memory that will eventually hold the array elements has not been allocated yet. This is stage two.

What information do you think would be required in order to reserve enough space in the computer's memory for all the array elements?

Well, it would be necessary to state the size of the array, that is the *maximum* number of elements required by the array. Also, since each data type requires a different amount of memory space, it is also necessary to state the type of each individual array element (this will be the same type used in stage one of the array declaration). The array type and size are then put together with a special **`new`** operator. For example, if we required an array of 10 integers the following would be appropriate:

```
someArray = new int[10];
```

The **`new`** operator creates the space in memory for an array of the given size and element type.[1]

[1] Of course this size should not be a negative value. A negative value will cause an error in your program.

We will come back to look at this **new** operator when looking at classes and objects in the next chapter. Once the size of the array is set it cannot be changed, so always make sure you create an array that is big enough for your purpose. Returning to the temperature example above, if you wanted the array to hold seven temperatures you would allocate memory as follows:

```
temperature = new double[7];
```

Let's see what effect the **new** operator has on computer memory by looking at figure 6.2.

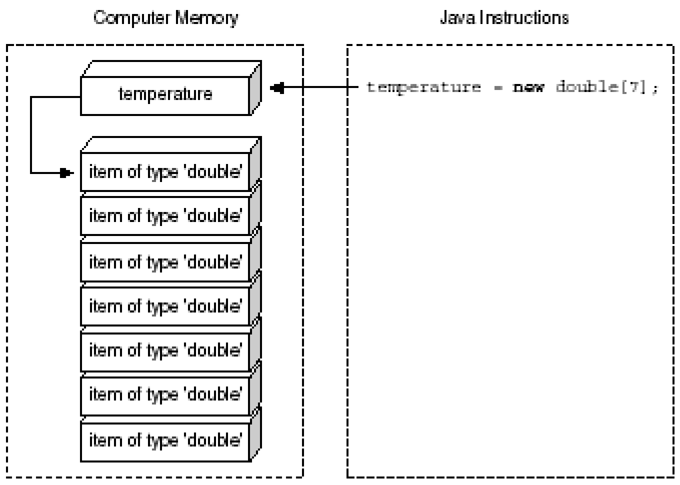


Fig 6.2. The effect on computer memory of declaring an array of seven 'double' values

As can be seen from figure 6.2, the array creation has created seven continuous locations in memory. Each location is big enough to store a value of type **double**. The `temperature` variable is linked to these seven elements by containing the address of the very first element. In effect, the array reference, `temperature`, is linked to seven new variables. Each of these variables will automatically have some initial value placed in them. If the variables are of some number type (such as **int** or **double**) the value of each will be initially set to zero; if the variables are of type **char** their values will be set initially to a special Unicode value that represents an empty character; if the variables are of **boolean** type they will each be set initially to **false**.

The two stages of array creation (declaring and allocating memory space for the elements) can also be combined into one step as follows:

```
double[] temperature = new double[7];
```

You may be wondering: what names have each of these variables been given? The answer is that each element in an array shares the same name as the array, so in this case each element is called `temperature`. The individual elements are then *uniquely identified* by an additional **index value**. An index value acts rather like a street number to identify houses on the same street (see figure 6.3). In much the same way as a house on a street is identified by the street name and a house number, an array element is identified by the array name and the index value.

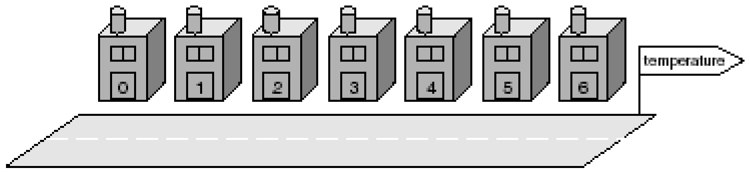


Fig 6.3. Elements in an array are identified in much the same way as houses on a street

Like a street number, these index values are always contiguous integers. Note carefully that, in Java, *array indices start from 0 and not from 1*. This index value is always enclosed in square brackets**,** so the first temperature in the list is identified as `temperature[0]`, the second temperature by `temperature[1]` and so on.

This means that the size of the array and the last index value are *not* the same value. In this case the size is 7 and the last index is 6. There is no such value as `temperature[7]`, for example. Remember this, as it is a common cause of errors in programs! If you try and access an invalid element such as `temperature[7]`, the following program error will be generated by the system:

```
java.lang.ArrayIndexOutOfBoundsException
```

This type of error is called an *exception*. You will find out more about exceptions in the second semester but you should be aware that, very often, exceptions will result in program termination.

Usually, when an array is created, values will be added into it as the program runs. If, however, all the values of the array elements are known beforehand, then an array can be created without the use of the **`new`** operator by initializing the array as follows:

```
double[] temperature = {9, 11.5, 11, 8.5, 7, 9, 8.5} ;
```

The initial values are placed in braces and separated by commas. The compiler determines the length of the array by the number of initial values (in this case 7). Each value is placed into the array in order, so `temperature[0]` is set to 9, `temperature[1]` to 11.5 and so on. This is the only instance in which *all the elements* of an array can be assigned explicitly by listing out the elements in a single assignment statement.

Once an array has been created, elements must be accessed *individually*.

6.3 Accessing array elements

Once an array has been created, its elements can be used like any other variable of the given type in Java. If you look back at the temperature example, initializing the values of each temperature when the array is created is actually quite unrealistic. It is much more likely that temperatures would be entered into the program as it runs. Let's look at how to achieve this.

Whether an array is initialized or not, values can be placed into the individual array elements. We know that each element in this array is a variable of type **`double`**. As with any variable of a primitive type, the assignment operator can be used to enter a value.

The only thing you have to remember when using the assignment operator with an array element is to specify *which* element to place the value in. For example, to allow the user of the program to enter the value of the first temperature, the following assignment could be used (again, assuming the existence of a `Scanner` object, `keyboard`):

```
temperature[0] = keyboard.nextDouble();
```

Note again that, since array indices begin at 0, the first temperature is not at index 1 but index 0.

Array elements could also be printed on the screen. For example, the following command prints out the value of the *sixth* array element:

```
System.out.println(temperature[5]); // index 5 is the sixth element!
```

Note that an array index (such as 5) is just used to *locate* a position in the array; it is *not* the item at that position.

For example, assume that the user enters a value of 25.5 for the first temperature in the array; the following statement:

```
System.out.println("temperature for day 1 is "+ temperature[0]);
```

would then print out the message:

```
temperature for day 1 is 25.5
```

Statements like the `println` command above might seem a bit confusing at first. The message refers to "temperature for day **1**" but the temperature that is displayed is `temperature[0]`. Remember though that the temperature at index position 0 *is* the first temperature! After a while you will get used to this indexing system of Java.

As you can see from the examples above, you can use array elements in exactly the same way you can use any other kind of variable of the given type. Here are a few more examples:

```
temperature[4] = temperature[4] * 2;
```

This assignment doubles the value of the *fifth* temperature. The following **`if`** statement checks if the temperature for the *third* day was a hot temperature:

```
if (temperature[2] >= 18)
{
   System.out.println("it was hot today");
}
```

So far so good, but if you are just going to use array elements in the way you used regular variables, why bother with arrays at all?

The reason is that the indexing system of arrays is in fact a very powerful programming tool. The index value does not need to be a literal number such as 5 or 2 as in the examples we have just shown you; it can be *any expression that evaluates to an integer*.

More often than not an integer *variable* is used, in place of a fixed index value, to access an array element. For example, if we assume that `i` is some integer variable, then the following is a perfectly legal way of accessing an array element:

```
System.out.println(temperature[i]); // index is a variable
```

Here the array index is not a literal number (like 2 or 5) but the variable `i`. The value of `i` will determine the array index. If the value of `i` is 4 then this will display `temperature[4]`, if the value of `i` is 6 then this will display `temperature[6]`, and so on. One useful application of this is to place the array instructions within a loop (usually a **`for`** loop), with the loop counter being used as the array index. For example, returning to the original problem of entering all seven temperature readings, the following loop could now be used:

```
for(int i = 0; i<7; i++) // note, loop counter runs from 0 to 6
{
   System.out.println("enter max temperature for day "+(i+1));
   temperature[i] = keyboard.nextDouble(); // use loop counter
}
```

Note carefully the following points from this loop:

- Unlike many of the previous examples of **`for`** loop counters that started at 1, this counter starts at 0. Since the counter is meant to track the array indices, 0 is the appropriate number to start from.
- The counter goes up to, but does not include, the number of items in the array. In this case this means the counter goes up to 6 and not 7. Again this is because the array index for an array of size 7 stops at 6.

- The `println` command uses the loop counter to display the number of the given day being entered. The loop counter starts from 0, however. We would not think of the first day of the week as being day 0! In order for the message to be more meaningful for the user, therefore, we have displayed (`i+1`) rather than `i`.

Effectively the following statements are executed by this loop:

```
System.out.println("enter max temperature for day 1 ");   1st time round loop
temperature[0] = keyboard.nextDouble();

System.out.println("enter max temperature for day 2 ");   2nd time round loop
temperature[1] = keyboard.nextDouble();

//as above but with indices 2-5                            3rd–6th time round loop

System.out.println("enter max temperature for day 7 ");   7th time round loop
temperature[6] = keyboard.nextDouble();
```

You should now be able to see the benefit of an array. This loop can be made more readable if we make use of a built-in feature of all arrays that returns the length of an array. It is accessed by using the word `length` after the name of the array. The two are joined by a full stop. Here is an example:

```
System.out.print("number of temperatures = ");
System.out.println(temperature.length); // returns the size of the array
```

which displays the following on to the screen:

```
number of temperatures = 7
```

Note that `length` feature returns the size of the array, not necessarily the number of items currently stored in the array (which may be fewer). This attribute can be used in place of a fixed number in the **`for`** loop as follows:

```
for (int i = 0; i < temperature.length, i++)
{
   // code for loop goes here
}
```

To see this technique being exploited, look at program 6.1 to see the completed `TemperatureReadings` program, which stores and displays the maximum daily temperatures in a week.

Program 6.1

```
import java.util.*;
public class TemperatureReadings
```

```
{
   public static void main(String[] args)
   {
        Scanner keyboard = new Scanner(System.in);
        // create array
        double[] temperature;
        temperature = new double[7];
        // enter temperatures
        for (int i = 0; i < temperature.length; i++)
        {
           System.out.println("enter max temperature for day " + (i+1));
           temperature[i] = keyboard.nextDouble();
        }
        // display temperatures
        System.out.println(); // blank line
        System.out.println("***TEMPERATURES ENTERED***");
        System.out.println(); // blank line
        for (int i = 0; i < temperature.length; i++)
        {
           System.out.println("day "+(i+1)+" "+ temperature[i]);
        }
   }
}
```

Note how `length` was used to control the two **`for`** loops. Here is a sample test run.

```
enter max temperature for day 1 12.2
enter max temperature for day 2 10.5
enter max temperature for day 3 13
enter max temperature for day 4 15
enter max temperature for day 5 13
enter max temperature for day 6 12.5
enter max temperature for day 7 12

***TEMPERATURES ENTERED***

day 1 12.2
day 2 10.5
day 3 13.0
day 4 15.0
day 5 13.0
day 6 12.5
day 7 12.0
```

6.4 Passing arrays as parameters

In chapter 5 we looked at how methods can be used to break up a programming task into manageable chunks. Methods can receive data in the form of parameters and can send back data in the form of a return value. Arrays can be used both as parameters to methods and as return values. In the next section we will see an example of an array as a return value from a method. In this section we will look at passing arrays as parameters to a method. As an example of passing an array to a method, consider once again program 6.1, which processes temperature readings. That program contains all the processing within the `main` method. As a result, the code for this method is a little difficult to read. Let's do something about that. We will create two methods, `enterTemps` and `displayTemps`, to enter and display temperatures respectively. To give these methods access to the array they must receive it as a parameter. Here, for example, is the header for the `enterTemps` method. Notice that when a parameter is declared as an array type, the size of the array is not required but the empty square brackets are:

```
static void enterTemps( double[] temperatureIn )
{
   // rest of method goes here
}
```

Now, although in the previous chapter we told you that a parameter just receives a copy of the original variable, this works a little differently with arrays. We will explain this a little later, but for now just be aware that this method will actually fill the original array. The code for the method itself is straightforward:

```
Scanner keyboard = new Scanner(System.in); // create local Scanner object
for (int i = 0; i < temperatureIn.length; i++)
{
   System.out.println("enter max temperature for day " + (i+1));
   temperatureIn[i] = keyboard.nextDouble();
}
```

Similarly the `displayTemps` method will require the array to be sent as a parameter. Program 6.2 rewrites program 6.1 by adding the two methods mentioned above. Take a look at it and then we will discuss it.

Program 6.2

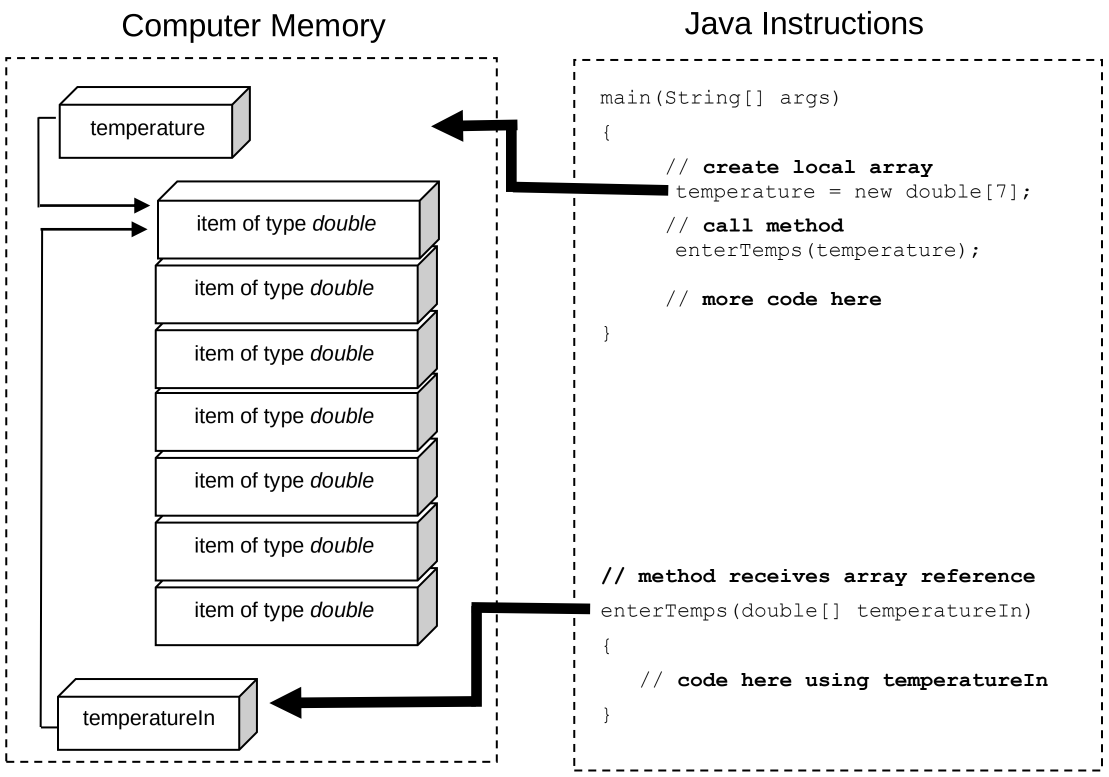
```
import java.util.*;
public class TemperatureReadings2
{
     public static void main(String[] args)
     {
        double[] temperature;
        temperature = new double[7];
        enterTemps(temperature); // call method
        displayTemps(temperature); // call method
     }
     // method to enter temperatures
     static void enterTemps(double[] temperatureIn)
     {
        Scanner keyboard = new Scanner(System.in);
        for (int i = 0; i < temperatureIn.length; i++)
        {
          System.out.println("enter max temperature for day " + (i+1));
          for temperatureIn[i] = keyboard.nextDouble();
        }
     }
     // method to display temperatures
     static void displayTemps(double[] temperatureIn)
     {
      System.out.println();
      System.out.println("***TEMPERATURES ENTERED***");
      System.out.println();
      for (int i = 0; i < temperatureIn.length; i++)
      {
        System.out.println("day "+(i+1)+" "+ temperatureIn[i]);
      }
    }
}
```

Notice that when sending an array as a parameter, the array name alone is required

```
public static void main(String[] args)
{
   double[] temperature;
   temperature = new double[7];
   enterTemps(temperature); // array name plugged in
   displayTemps(temperature); // array name plugged in
}
```

Now let us return to the point we made earlier. You are aware that, in the case of a simple variable type such as an **`int`**, it is the *value* of the variable that is copied when it is passed as a parameter. This means that if the value of a parameter is altered within a method, the original variable is unaffected outside that method. This works differently with arrays.

As we said earlier, the `enterTemps` method actually fills the *original* array. How can this be? The answer is that in the case of arrays, the value sent as a parameter is not a copy of each array element but, instead, a copy of the array *reference*. In other words, the *location* of the array is sent to the receiving method not the value of the contents of the array. Now, even though the receiving parameter (`temperatureIn`) has a different name to the original variable in `main` (`temperature`), they are both pointing to the same place in memory so both are modifying the same array! This is illustrated in figure 6.3: Sending the array reference to a method rather than a copy of the whole array is a very efficient use of the computer's resources, especially when arrays become very large.



```
main(String[] args)
{
     // create local array
      temperature = new double[7];
     // call method
      enterTemps(temperature);

     // more code here
}

// method receives array reference
enterTemps(double[] temperatureIn)
{
   // code here using temperatureIn
}
```

Fig 6.3. The effect on computer memory of passing an array as a parameter

6.5 Returning an array from a method

A method can return an array as well as receive arrays as parameters. As an example, let us reconsider the `enterTemps` method from program 6.2. At the moment, this method accepts an array as a parameter and fills this array with temperature values. Since this method fills the *original* array sent in as a parameter, it does not need to return a value – its return type is therefore **`void`**:

```
static void enterTemps(double[] temperatureIn)
{
    // code to fill the parameter, 'temperatureIn', goes here
}
```

An alternative approach would be *not* to send an array to this method but, instead, to create an array *within* this method and fill *this* array with values. This array can then be returned from the method:

```
// this method receives no parameter but returns an array of doubles
static double[] enterTemps()
{
    Scanner keyboard = new Scanner(System.in);
    // create an array within this method
    double[] temperatureOut = new double[7];
    // fill up the array created in this method
    for (int i = 0; i < temperatureOut.length; i++)
    {
        System.out.println("enter max temperature for day " + (i+1));
        temperatureOut[i] = keyboard.nextDouble();
    }
    // send back the array created in this method
    return temperatureOut;
}
```

As you can see, we use square brackets to indicate that an array is to be returned from a method :

```
static double[] enterTemps()
```

The array itself is created within the method. We have decided to call this array `temperatureOut`:

```
double[] temperatureOut = new double[7];
```

After the array has been filled it is sent back with a **`return`** statement. Notice that, to return an array, the name alone is required:

```
return temperatureOut;
```

Now that we have changed the `enterTemps` method, we need to revisit the original `main` method also. It will no longer compile now that the `enterTemps` method has changed:

```
// the original 'main' method will no loger compile!
public static void main(String[] args)
{
   double[] temperature = new double[7];
   enterTemps(temperature); // this line will now cause a compiler error !!
   displayTemps(temperature);
}
```

The call to `enterTemps` will no longer compile as the new `enterTemps` does not expect to be given an array as a parameter. The correct way to call the method is as follows:

```
enterTemps(); // this method requires no parameter
```

However, this method now *returns* an array. We really should do something with the array value that is returned from this method. We should use the returned array value to set the value of the original `temperature` array:

```
// just declare the 'temperature' array but do not allocate it memory yet
double[] temperature;
// 'temperature' array is now set to the return value of 'enterTemps'
temperature = enterTemps();
```

As you can see, we have not sized the `temperature` array once it has been declared. Instead the `temperature` array will be set to the size of the array returned by `enterTemps`, and it will contain all the values of the array returned by `enterTemps`. Program 6.3 presents the complete program:

Program 6.3

```
import java.util.*;
public class TemperatureReadings3
{
  public static void main(String[] args)
  {
    double[] temperature ;
    temperature = enterTemps(); // call new version of this method
    displayTemps(temperature);
  }
  // method to enter temperatures returns an array
  static double[] enterTemps()
  {
   Scanner keyboard = new Scanner(System.in);
   double[] temperatureOut = new double[7];
   for (int i = 0; i < temperatureOut.length; i++)
   {
       System.out.println("enter max temperature for day " + (i+1));
       temperatureOut[i] = keyboard.nextDouble();
   }
   return temperatureOut;
  }
  // this method is unchanged
  static void displayTemps(double[] temperatureIn)
  {
   System.out.println();
   System.out.println("***TEMPERATURES ENTERED***");
   System.out.println();
   for (int i = 0; i < temperatureIn.length; i++)
   {
      System.out.println("day "+(i+1)+" "+ temperatureIn[i]);
   }
  }
}
```

Program 6.3 behaves in exactly the same way as program 6.2, so whether you implement `enterTemps` as in program 6.2 or as in program 6.3 is really just a matter of preference.

6.6 The enhanced 'for' loop

As you can see from the examples above, when processing an entire array a loop is required. Very often, this will be a **`for`** loop. With a **`for`** loop, the loop counter is used as the array index within the body of the loop. In the examples above, the loop counter was used not only as an array index but also to display meaningful messages to the user. For example:

```
for (int i = 0; i < temperature.length; i++)
{
     System.out.println("day "+(i+1)+" "+ temperature[i]);
}
```

Here the loop counter was used to determine the day number to display on the screen, as well as the index of an array element. Very often, when a **`for`** loop is required, the *only* use made of the loop counter is as an array index to access all the elements of the array consecutively. Java provides an enhanced version of the **`for`** loop especially for this purpose.

Rather than use a counter, the enhanced **`for`** loop consists of a variable that, upon each iteration, stores consecutive elements from the array.[2] For example, if we wished to display on the screen each value from the `temperature` array, the enhanced **`for`** loop could be used as follows:

```
/* the enhanced for loop iterates through elements of an array without the need
   for an array index */

for (double item : temperature) // see discussion below
{
   System.out.println(item);
}
```

In this case we have named each successive array element as `item`. The loop header is to be read as "for each `item` in the `temperature` array". For this reason the enhanced **`for`** loop is often referred to as the *for each* loop. Notice that the type of the array item also has to be specified. The type of this variable is **`double`** as we are processing an array of **`double`** values. Remember that this is the type of each *individual* element within the array.

You should note that the variable `item` can be used only within the loop, we cannot make reference to it outside the loop. Within the body of the loop we can now print out an array element by referring directly to the `item` variable rather than accessing it via an index value:

[2] The enhanced **`for`** loop also works with other classes in Java, which act as alternatives to arrays. We will explore some of these classes in chapter 15.

```
System.out.println(item); // 'item' is an array element
```

This is a much neater solution than using a standard **`for`** loop, which would require control of a loop counter in the loop header, and array look up within the body of the loop.

Be aware that the enhanced **`for`** loop should *not* be used if you wish to modify the array items. Modifying array items with such a loop will not cause a compiler error, but it is unsafe as it may cause your program to behave unreliably. So you should use an enhanced **`for`** loop only when

- you wish to access the *entire* array (and not just part of the array);
- you wish to *read* the elements in the array, not *modify* them;
- you do not require the array index for additional processing.

Very often, when processing an array, it is the case that these three conditions apply. In the following sections we will make use of this enhanced **`for`** loop where appropriate.

6.7 Some useful array methods

Apart from the `length` feature, an array does not come with any useful built in routines. So we will develop some of our own methods for processing an array. We will use a simple integer array for this example. Here is the outline of the program we are going to write in order to do this:

```
import java.util.*;
public class SomeUsefulArrayMethods
{
   public static void main (String[] args)
   {
      Scanner keyboard = new Scanner(System.in);
      int[] someArray; // declare an integer array
      // ask user to determine size of array
      System.out.println("How many elements to store?");
      int size = keyboard.nextInt();
      // size array now
      someArray = new int[size];

      // call methods here
   }
   // methods to process an array here
}
```

As you can see, we have delayed the second stage of array creation here until the user tells us how many elements to store in the array. Now to some methods.

6.7.1 Array maximum

The first method we will develop will allow us to find the maximum value in an array. For example, we may have a list of scores and wish to know the highest score in this list. Finding the maximum value in an array is a much better approach than the one we took in chapter 5, where we looked at a method to find the maximum of two values and another method to find the maximum of three values. This array method can instead be used with lists of two, three, four or any other number of values. The approach we will use will be similar to the `max` method we developed in chapter 5 for finding the maximum of three values. Here is the pseudocode again.

```
SET result TO first number
IF second number > result
BEGIN
   SET result TO second number
END
IF third number > result
BEGIN
   SET result TO third number
END
RETURN result
```

Here, the final result is initialized to the first value. All other values are then compared with this value to determine the largest value. Now that we have an array, we can use a loop to process this comparison, rather than have a series of many **`if`** statements. Here is a suitable algorithm:

```
SET result TO first value in array
LOOP FROM second element in array TO last element in array
BEGIN
   IF current element > result
   BEGIN
     SET result TO current element
   END
END
RETURN result
```

This method will need the array that it has to search to be sent in as a parameter. Also, this method will return the maximum item so it must have an integer return type.

```
static int max (int[] arrayIn)
{
   int result = arrayIn[0]; // set result to the first value in the array
   // this loops runs from the 2nd item to the last item in the array
   for (int i=1; i < arrayIn.length; i++)
   {
      if (arrayIn[i] > result)
      {
         result = arrayIn[i]; // reset result to new maximum
      }
   }
   return result;
}
```

Notice we did not use the enhanced **`for`** loop here, as we needed to iterate from the *second* item in the array rather than through *all* items, and the standard **`for`** loop gives us this additional control.

6.7.2 Array summation

The next method we will develop will be a method that calculates the total of all the values in the array. Such a method might be useful, for example, if we had a list of deposits made into a bank account and wished to know the total value of these deposits. A simple way to calculate the sum is to keep a running total and add the value of each array element to that running total. Whenever you have a running total it is important to initialize this value to zero. We can express this algorithm using pseudocode as follows:

```
SET total TO zero
LOOP FROM first element in array TO last element in array
BEGIN
    SET total TO total + value of current element
END
RETURN total
```

This method will again need the array to be sent in as a parameter, and will return an integer (the value of the sum), giving us the following:

```
static int sum (int[] arrayIn)
{
    int total = 0;
    for (int currentElement : arrayIn)
    {
        total = total + currentElement;
    }
    return total;
}
```

Notice the use of the enhanced **`for`** loop here – as we need to iterate through *all* elements within the array.

6.7.3 Array membership

It is often useful to determine whether or not an array contains a particular value. For example, if the list were meant to store a collection of unique student ID numbers, this method could be used to check if a new ID number already exists before adding it to the list. A simple technique is to check each item in the list one by one, using a loop, to see if the given value is present. If the value is found the loop is exited. If the loop reaches the end without exiting then we know the item is not present. Here is the pseudocode:

```
LOOP FROM first element in array TO last element in array
BEGIN
    IF current element = item to find
    BEGIN
        EXIT loop and RETURN true
    END
END
RETURN false
```

Notice in the algorithm above that a value of **`false`** would be returned only if the given item is not found. If the value is found, the loop would terminate without reaching its end and a value of **`true`** would be returned.

Here is the Java code for this method. We need to ensure that this method receives the array to search and the item being searched for. Also the method must return a **`boolean`** value:

```
static boolean contains (int[] arrayIn, int valueIn)
{
    // enhanced 'for' loop used here
    for (int currentElement : arrayIn)
    {
        if (currentElement == valueIn)
        {
            return true; // exit loop early if value found
        }
    }
    return false; // value not present
}
```

6.7.4 Array search

One of the most common tasks relating to a list of values is to determine the position of an item within the list. For example, we may wish to know the position of a job waiting in a printer queue.

How will we go about doing this?

Just as we did when devising the `contains` algorithm, we will need to use a loop to examine every item in the array. Inside the loop we check each item one at a time and compare it to the item we are searching for.

What do we do if we find the item we are searching for? Well, we are interested in its position in the array so we just return the index of that item.

Now we need to decide what we do if we reach the end of the loop, having checked all the elements in the array, without finding the item we are searching for. This method needs to return an integer regardless of whether or not we find an item. What number shall we send back if the item is not found? We need to send back a value that could never be interpreted as an array index. Since array indexes will always be positive numbers we could send back a negative number, such as -999, to indicate a valid position has not been found.

Here is the pseudocode:

```
LOOP FROM first element in array TO last element in array
BEGIN
    IF current element = item to find
    BEGIN
      EXIT loop and RETURN current index
    END
END
RETURN -999
```

This approach is often referred to as a *linear search*. Here is the Java code for this method. Once again we need to ensure that this method receives the array to search and the item being searched for. This method must return an integer value:

```
static int search (int[] arrayIn, int valueIn)
{
   // enhanced 'for' loop should not be used here!
   for (int i=0; i < arrayIn.length; i++)
   {
      if (arrayIn[i] == valueIn)
      {
         return i; // exit loop with array index
      }
   }
   return -999; // indicates value not in list
}
```

Notice, in this case, we could not use the enhanced **`for`** loop. The reason for this is that we required the method to return the array index of the item we are searching for. This index is best arrived at by making use of a loop counter in a standard **`for`** loop.

6.7.5 The final program

The complete program for manipulating an array is now presented below. The array methods are accessed via a menu. We have included some additional methods here for entering and displaying an array:

Program 6.4

```
import java.util.*;

// a menu driven program to test a selection of useful array methods

public class SomeUsefulArrayMethods
{
   public static void main (String[] args)
   {
     char choice;
     Scanner keyboard = new Scanner(System.in);
     int[] someArray; // declare an integer array
     System.out.print("How many elements to store?: ");
     int size = keyboard.nextInt();
     // size the array
     someArray = new int [size];
     // menu
     do
     {
          System.out.println();
          System.out.println("[1] Enter values");
          System.out.println("[2] Find maximum");
          System.out.println("[3] Calculate sum");
          System.out.println("[4] Check membership");
          System.out.println("[5] Search array");
          System.out.println("[6] Display values");
          System.out.println("[7] Exit");
          System.out.print("Enter choice [1-7]: ");
          choice = keyboard.next().charAt(0);
          System.out.println();
          // process choice by calling additional methods
          switch(choice)
```

```
        {
           case '1': fillArray(someArray);
                     break;
           case '2': int max = max(someArray);
                     System.out.println("Maximum array value = " + max); break;
           case '3': int total = sum(someArray);
                     System.out.println("Sum of array values = " + total); break;
           case '4': System.out.print ("Enter value to find: ");
                     int value = keyboard.nextInt();
                     boolean found = contains(someArray, value);
                     if (found)
                     {
                        System.out.println(value + " is in the array");
                     }
                     else
                     {
                        System.out.println(value + " is not in the array");
                     }
                     break;
           case '5': System.out.print ("Enter value to find: ");
                     int item = keyboard.nextInt();
                     int index = search(someArray, item);
                     if (index == -999) // indicates value not found
                     {
                        System.out.println ("This value is not in the array");
                     }
                     else
                     {
                        System.out.println ("This value is at array index " + index);
                     }
                     break;
           case '6': System.out.println("Array values");
                     displayArray(someArray);
                     break;
        }
    } while (choice != '7');
    System.out.println("Goodbye");
}
// additional methods

// fills an array with values
static void fillArray(int[] arrayIn)
{
    Scanner keyboard = new Scanner (System.in);
    for (int i = 0; i < arrayIn.length; i++)
    {
       System.out.print("enter value ");
      arrayIn[i] = keyboard.nextInt();
    }
}

// returns the total of all the values held within an array
static int sum (int[] arrayIn)
{
   int total = 0;
   for (int currentElement : arrayIn)
   {
      total = total + currentElement;
   }
   return total;
}

// returns the maximum value in an array
 static int max (int[] arrayIn)
 {
     int result = arrayIn[0]; // set result to the first value in the array
     // this loops runs from the 2nd item to the last item in the array
     for (int i=1; i < arrayIn.length; i++)
     {
     if (arrayIn[i] > result)
     {
        result = arrayIn[i]; // reset result to new maximum
     }
   }
   return result;
}
```

```
    // checks if a given item is contained within the array
    static boolean contains (int[] arrayIn, int valueIn)
    {
       // enhanced 'for' loop used here
       for (int currentElement : arrayIn)
       {
          if (currentElement == valueIn)
          {
             return true; // exit loop early if value found
          }
       }
       return false; // value not present
    }


    /* returns the position of an item within an array or -999 if the value is not present within the
       array */
    static int search (int[] arrayIn, int valueIn)
    {
      for (int i = 0; i < arrayIn.length; i++)
      {
       if (arrayIn[i] == valueIn)
       {
         return i;
       }
      }
      return -999;
    }

    // displays the array values on the screen
    static void displayArray(int[] arrayIn)
    {
     System.out.println();
     // standard 'for' loop used here as the array index is required
     for (int i = 0; i < arrayIn.length; i++)
     {
         System.out.println("array[" + i + "] = " + arrayIn[i]);
     }
    }
}
```

Here is a sample program run:

```
How many elements to store?: 5


[1] Enter values
[2] Find maximum
[3] Calculate sum
[4] Check membership
[5] Search array
[6] Display values
[7] Exit
Enter choice [1-7]: 1


enter value 12
enter value 3
enter value 7
enter value 6
enter value 2


[1] Enter values
[2] Find maximum
[3] Calculate sum
[4] Check membership
[5] Search array
[6] Display values
[7] Exit
Enter choice [1-7]: 2


Maximum array value = 12

[1] Enter values
[2] Find maximum
[3] Calculate sum
[4] Check membership
[5] Search array
[6] Display values
[7] Exit
Enter choice [1-7]: 3


Sum of array values = 30

[1] Enter values
[2] Find maximum
[3] Calculate sum
[4] Check membership
[5] Search array
[6] Display values
[7] Exit Enter choice [1-7]: 4


Enter value to find: 10
```

```
10 is not in the array

[1] Enter values
[2] Find maximum
[3] Calculate sum
[4] Check membership
[5] Search array
[6] Display values
[7] Exit
Enter choice [1-7]: 4

Enter value to find: 7

7 is in the array

[1] Enter values
[2] Find maximum
[3] Calculate sum
[4] Check membership
[5] Search array
[6] Display values
[7] Exit
Enter choice [1-7]: 5


Enter value to find: 7
This value is at array index 2

[1] Enter values
[2] Find maximum
[3] Calculate sum
[4] Check membership
[5] Search array
[6] Display values
[7] Exit
Enter choice [1-7]: 6
```

```
Array values

array[0] = 12
array[1] = 3
array[2] = 7
array[3] = 6
array[4] = 2


[1] Enter values
[2] Find maximum
[3] Calculate sum
[4] Check membership
[5] Search array
[6] Display values
[7] Exit
Enter choice [1-7]: 7


Goodbye
```

Self-test questions

1 When is it appropriate to use an *array* in a program?

2 Consider the following explicit creation of an array:

```
int[] someArray = {2,5,1,9,11};
```

a) What would be the value of `someArray.length` ?

b) What is the value of `someArray[2]`?

c) What would happen if you tried to access `someArray[6]`?

d) Create the equivalent array by using the **`new`** operator and then assigning the value of each element individually.

e) Write a standard **`for`** loop that will double the value of every item in `someArray`.

f) Explain why, in this example, it would not be appropriate to use an enhanced **`for`** loop.

g) Use an enhanced **`for`** loop to display the values inside the array.

h) Modify the enhanced **`for`** loop above so that only numbers greater than 2 are displayed.

Programming exercises

1 Copy program 6.4 (`SomeUsefulArrayMethods`), which manipulates an array of integers, then add additional methods to the program in order to

a) return the average from the array of integers (make use of the `sum` method to help you calculate the average);

b) display on the screen all those values greater than or equal to the average.