Chapter 20
Arrays

20.1 Introduction

This chapter discusses how arrays are represented in Scala. Arrays are (logically) a continuous set of slots that can hold values or references to instances of a specific type.

20.2 Arrays

Arrays in Scala are objects, like most other data types. Like arrays in any other language, they hold elements of data in an order specified by an index. They are Zero-based arrays, as in C, which means that an array with 10 elements is indexed from 0 to 9.

To create a new array, you must specify the type of array object and the number of elements in the array. The Array type defines *array* like behaviour in Scala. The type of element held in the Array is indicated within a set of square brackets after the Array. The number of elements is specified by an integer between round brackets. As an array is an instance, it is created in the usual way using the new operation:

```
new Array[String](10);
```

This creates an array capable of holding ten String objects. We can assign such an array instance to a variable by specifying that the variable holds an array. You do this by indicating the type of the array to be held by the variable along with the array indicator. Notice that we do not specify the number of array locations that are held by the array variable:

```
val myArray: Array[String]
var names: Array[String]
```


J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_20

Both of the above define variables which can hold a reference to an array of Strings. However, neither declaration defines how large the array object being referenced will be. This is not a problem as myArray and names will only hold a reference to (or an address of) an array. The reference is the same size whatever the length of array being pointed to.

We now create an array and assign it to our variable:

```
val myArray: Array[String] = new Array[String](3)
```

We could also have used the shorter from relying on type inference for the myArray val:

```
val myArray = new Array[String](3)
```

There is a short cut way to create and initialise an array:

```
val myArray = Array("John", "Denise", "Phoebe", "Adam")
```

This relies on both Scala to infer the type of the myArray val and for Scala to use a factory method (called apply) to construct the array. You can use the apply factory method directly if you wish, and thus the above is semantically the same as:

```
val myArray2 = Array.apply("John", "Denise", "Phoebe", "Adam")
```

Both of the above create an array of four elements containing the strings “John”, “Denise”, “Phoebe” and “Adam”. We can change any of these fields by specifying the appropriate index and replacing the existing value with a new string:

```
val myArray = Array.apply("John", "Denise", "Phoebe", "Adam")
myArray(3) = "Isobel"
```

The above statement replaces the string “Adam” with the string “Isobel”. Merely being able to put values into an array would be of little use; we can also access the array locations in a similar manner:

```
myArray.apply(0)
```

This retrieves the current value held in the array referenced by myArray. Once again this is a common enough operation that a shorthand (and more commonly used variant) is available:

```
myArray(0)
```

This can be used to retrieve the *zeroth* value in the array. For example,

```
println("The name in position 1 is " + myArray(0));
```

The above statement results in the following string being printed:

```
The name in position 1 is John
```

As arrays are objects, we can also obtain information from them. For example, to find out how many elements are in the array we can use the instance variable `length`:

```
myArray.length
```

Arrays are fixed in length when they are created, whereas vectors can change their length. To obtain the size of an array, you can access instance variable, `length`, but you must use a method, `size`, to determine the current size of a vector.

Arrays can be passed into and out of methods very simply by specifying the type of the array, the name of the variable to receive the array and the array indicator.

20.2.1 Arrays of Objects

The above examples have focussed on arrays of Strings; however, you can also create arrays of any type of object but this process is a little more complicated (it is actually exactly the same for strings, but some of what is happening is hidden from you). For example, assuming we have a class Person, then we can create an array of Persons:

```
val pa: Array[Person] = new Array[Person](4)
```

Figure 20.1 illustrates the result of creating an array of Person instances. It is important to realise what this gives you. It provides a variable *pa* that can hold a reference to an array object of Persons. At present this array is empty and *does not*

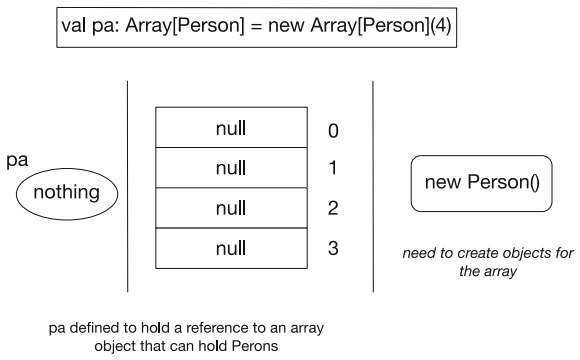
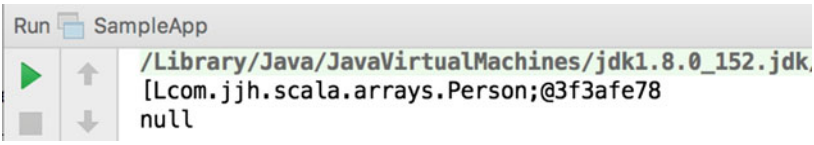


Fig. 20.1 Creating an array of objects

hold references to any instances of Person. For example, if you now print out the value of pa and the value of pa(0), that is,

```
object SampleApp extends App {
  val pa: Array[Person] = new Array[Person](4)
  println(pa)
  println(pa(0))
}
```

you will get:



Note that this indicates that the array is actually an array of references to the instances "held" in the array as opposed to an array of those instances. This is illustrated in the first part of Fig. 20.1. To actually make it hold instances of Person we must add each person instance to the appropriate array location. For example,

```
case class Person(name: String)

object SampleApp extends App {
  val pa: Array[Person] = new Array[Person](4)

  pa(0) = Person("John")
  pa(1) = Person("Denise")
  pa(2) = Person("Phoebe")
  pa(3) = Person("Adam")

  println(pa)
  println(pa(0))
}
```

This is illustrated in the last part Figs. 20.1 and 20.2. Thus the creation of an array of objects is a three-stage process:

1. Create a variable that can reference an array of the appropriate type of object.
2. Create the array object.
3. Fill the array object with instances of the appropriate type.

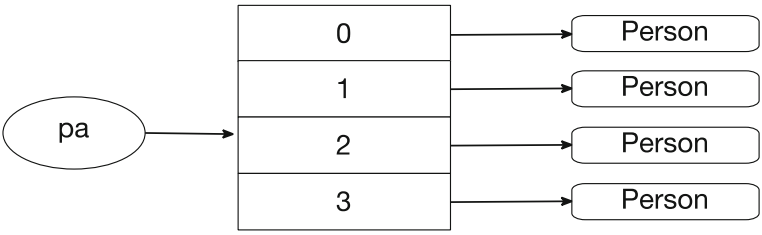


Fig. 20.2 Complete array structure

20.2.2 Ragged Arrays

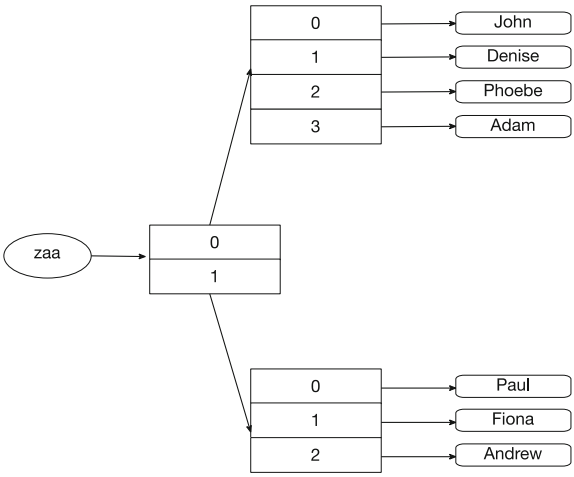
As in most high-level languages multidimensional arrays can be defined in Scala. This is done in the following manner:

```
val fa = Array(Array("John", "Denise", "Phoebe", "Adam"),
  Array("Paul", "Fiona", "Andrew"))
```

Pictorially this can be viewed as shown in Fig. 20.3.

As can be seen from this example, two-dimensional arrays in Scala are actually Array objects holding references to other Array objects. Thus in this case, the first array is an array of two elements (0 and 1). The type of these elements is an Array of Strings. The first array element 0 holds a reference to another array object (one that holds Strings), etc. This means that the structure created can be ragged—that is the second dimension in this example is not the same across the two subarrays: one is of length 4, and one is of length 3.

Fig. 20.3 A ragged Array



Thus if you create these arrays and object for the Person class, then we would:

1. Define the val *paa* as hold a reference to an array of arrays

   ```
   val paa: Array[Array[Person]]
   ```

2. Create the multidimensional array

   ```
   val paa: Array[Array[Person]] =
            new Array[Array[Person]](2)
   ```

   Note we have to specify the first dimension as it is necessary to allocate enough space for the required references. We do not have to specify the second dimension as these can be specified in the subsequent array object creation messages.

3. Create the subarrays:

   ```
   pa(0) = new Array[Person](4)
   pa(1) = new Array[Person](3)
   ```

4. We are now ready to add instances to the two-dimensional array, for example

   ```
   pa(0)(0) = new Person("John")
   ```

As you can see from this last example, multidimensional arrays are accessed in exactly the same way as single-dimensional arrays with one index following another (note each is within its own set of round brackets—()). That is you can access this two-dimensional array by specifying a particular position within the array using the same format:

```
println(matrix(2)(2))
```

Note that the way that multidimensional arrays are implemented in Scala means that you can easily implement any number of dimensions required, for example

```
val zaa = Array(
            Array(
              Array("John", "Denise", "Phoebe", "Adam"),
              Array("Paul", "Fiona", "Andrew")),
            Array (
              Array("Darren", "Val")))
```

This defines a three-dimensional array structure containing two 2-dimensional arrays. This is illustrated in Fig. 20.4.

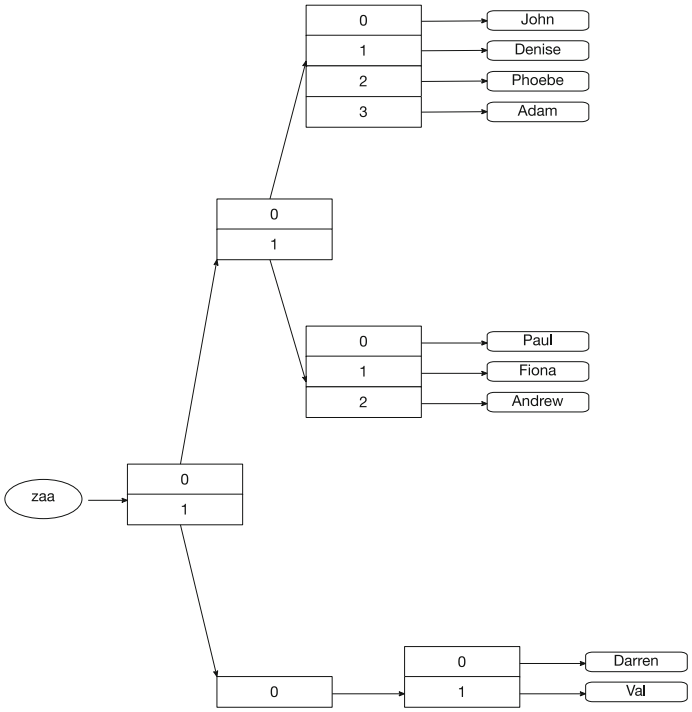


Fig. 20.4 A multidimensional ragged array

20.3 Creating Square Arrays

Of course, although it is possible to create raged arrays in Scala, it is far more common to create square arrays, for example a 2 by 2 or a 3 by 3 array. To do this the Array type provides a factory method *ofDim* that can be used to create an array of an appropriate dimension, for example

```
val array = Array.ofDim[Int](3, 2)
```

This creates a two-dimensional array of size 3 by 2 that can hold Int types. These following statements

```
println(array.length)
println(array(0).length)
println(array(1).length)
println(array(2).length)
```

produce this output

```
3
2
2
2
```

Of course this populates the array of Ints with a set of Zeros. If you wish to initialise such an array with other default values, then you can use the fill factory method on the Array type:

```
val myArray = Array.fill(2, 2, 2)(2.0)
println(myArray(0)(0)(0))

produces

2.0
```

This creates a three-dimensional array with all the elements of the array populated with the double value 2.0.

20.4 Looping Through Arrays

You can use a loop to process the contents of an array. For example, given an array, myArray, containing three strings:

```
val myArray = Array("Zero ", "One ", "Two")
```

We can loop through the elements in that array in a number of ways. For example, we can use the length of the array and an index to access each element in turn, for example

```
for (i <- 0 to myArray.length -1)
      println(myArray(i))
```

In this example the loop variable 'I' ranges from 0 to one less than the length of the array. This is because the array contains three strings, with indexes, Zero, one and two! This is important, if you try and access the element with the index '3' (which is the fourth element in the array) you will get an index out of bounds exception generated—that is your program will generate an error!

The result of running this program is

```
Zero
One
Two
```

However, the problem with this approach is that you must remember to ensure that you loop from Zero to one less than the length of the array (otherwise you will not access the elements of the array you expect, and you will generate a runtime error). An alternative approach is to loop through each of the elements in turn applying a function to those elements. This is actually an example of functional programming and uses the *foreach* function defined on the Array type. For example,

```
object ArrayTest2 extends App {
  val myArray = Array("Zero ", "One ", "Two")
  myArray.foreach(x => println(x))
}
```

With this approach the *foreach* function takes element in turn and *binds* it to the variable x above. It then executes the body of the behaviour provided (in this case println(x)). The end result is the same as before:

```
Zero
One
Two
```

However, it is a better (and simpler) abstraction of processing each of the elements in the array.

20.5 The Main Method Revisited

At this point you are ready to review the parameter passed into the main class method. As a reminder, it always has the following format:

```
object MainTest {
   def main(args: Array[String]): Unit = {
      ...
   }
}
```

From this you can see that the parameter passed into the main method is an array of strings. This array holds any command line arguments passed into the program.

We now have enough information to write a simple program that parses the main method command line arguments:

```
object ParseInput {
  def main(args: Array[String]): Unit = {
    println(args.length)
    require(args.length > 0)
    for (i <- args) {
      println("Argument is " + i)
    }
  }
}
```

This is a very simple program but it provides the basics for a command line parser. Do not worry if you do not understand the syntax of the whole program; we cover `if` statements and `for` loops later in the book.

Arrays in Scala are passed into methods by value. However, as parameters only hold a reference to the objects they contain, if those objects are modified internally, the array outside the method is also modified. This can be the cause of extreme frustration when trying to debug programs. Arrays can also be returned from methods:

```
modifiers methodName: type (...)
def returnNames: Array[Person] () {
  ...
}
```

As an example of an array-based application, consider the following application `ArrayDemo`, which calculates the average of an array of numbers. This array is created in the ArrayDemo application and is passed into the `processArray` method as a parameter. This method is defined on the class ArrayProcessor. Within this method, the values of the array are added together and the total is divided by the number of elements in the array (i.e. its length):

```
class ArrayProcessor {
  def processArray(myArray: Array[Int]): Unit = {
    var total = 0
    var average = 0
    for (i <- myArray) total = total + i
    average = total / myArray.length
    println("The average was: " + average)
  }
}

object ArrayDemo extends App {
  val processor = new ArrayProcessor()
  val intArray = Array(1, 4, 7, 9)
  processor.processArray(intArray)
}
```