Chapter 3

Working with Arrays

Fixed-length Arrays

- if you're suppling it with values, `new` keyword is not needed
- use `()` instead of `[]` to access elements

```
val nums = new Array[Int](10)
 // An array of ten integers, all initialized with zero
val a = new Array[String](10)
 // A string array with ten elements, all initialized with null
val s = Array("Hello", "World")
// An Array[String] of length 2—the type is inferred
s(0) = "Goodbye"
// Array("Goodbye", "World")
```

Array Buffer

```
val b = ArrayBuffer[Int]()
 // Or new ArrayBuffer[Int]
 // An empty array buffer, ready to hold integers
```

Adding Values

`+=` operator

- add one element
- add multiple elements

`++=` operator

- add another array
- you can append any collection with the ++= operator

Traversing Arrays and Array Buffers

most of the time you can use the same code for both
defining range

- `until` is similar to `to` method, but excludes last value

```
for (i <- 0 until a.length)
```

- to visit every i-th element, use `by i` : `0 until a.length by 2`
- to visit elements starting from the end of the array: `0 until a.length by -1`

- finally, if you don't need the index, use for (elem <- a)

Transfoming Arrays

The for/yield loop creates a new collection of the same type as the original collection

```
val a = Array(2, 3, 5, 7, 11)
val result = for (elem <- a) yield 2 * elem
 // result is Array(4, 6, 10, 14, 22)
```

if you only want to process the elements that match a particular condition, use guard

```
for (elem <- a if elem % 2 == 0) yield 2 * elem
//alternatively, you can use
a.filter(_ % 2 == 0).map(2 * _)
// or even
a filter { _ % 2 == 0 } map { 2 * _ }
```

🚧 when using yield, the result is a new collection—the original collection is not affected.

❗ *some additional examples about removing elements from arrays are in additional notes*

Common Algorithms

- sum
  In order to use the sum method, the element type must be a numeric type: either
  an integral or floating-point type or BigInteger/BigDecimal.
- min and max
- sorted method sorts an array or array buffer and returns the sorted array or array buffer, *without modifying the original*
  - you can also supply comparison function , but you should use the sortWith method

```
 val bDescending = b.sortWith(_ > _) // ArrayBuffer(9, 7, 2, 1)
```

- You can sort an array, but **not an array buffer**, in place:

```
val a = Array(1, 7, 2, 9)
scala.util.Sorting.quickSort(a)
 // a is now Array(1, 2, 7, 9)
```

For the min, max, and quickSort methods, the element type **must have** a comparison operation . This is the case for numbers, strings, and other types with the Ordered trait .

- Finally, if you want to display the contents of an array or array buffer, the mkString method lets you specify the separator between elements. A second variant has parameters for the prefix and suffix. For example

```
a.mkString(" and ")
 // "1 and 2 and 7 and 9"
a.mkString("<", ",", ">")
 // "<1,2,7,9>"
```

Multidimensional Arrays

Works similar as in java. To construct an array, use `ofDim` method. To acess an alement, use two paris of parentheses. You can also make `ragged arrays` , with varying row lengths

```scala
val matrix = Array.ofDim[Double](3, 4) // Three rows, four columns
matrix(row)(column) = 42

val triangle = new Array[Array[Int]](10)
for (i <- triangle.indices)  triangle(i) = new Array[Int](i + 1)
```

Interpolating with Java

You can see info about this on page 43 of the book (p 59 in pdf)