Chapter 17

Working with Other Collections

Scala has a rich collection library. This chapter gives you a tour of the most commonly used collection types and operations, showing just the parts you will use most frequently. Chapter 24 will provide a more comprehensive tour of what's available, and Chapter 25 will show how Scala's composition constructs are used to provide such a rich API.

17.1 SEQUENCES

Sequence types let you work with groups of data lined up in order. Because the elements are ordered, you can ask for the first element, second element, 103rd element, and so on. In this section, we'll give you a quick tour of the most important sequences.

Lists

Perhaps the most important sequence type to know about is class List, the immutable linked-list described in detail in the previous chapter. Lists support fast addition and removal of items to the beginning of the list, but they do not provide fast access to arbitrary indexes because the implementation must iterate through the list linearly.

This combination of features might sound odd, but they hit a sweet spot that works well for many algorithms. The fast addition and removal of initial elements means that pattern matching works well, as described in Chapter 15. The immutability of lists helps you develop correct, efficient algorithms because you never need to make copies of a list.

Here's a short example showing how to initialize a list, and access its head and tail:

```
scala> val colors = List("red", "blue", "green")
colors: List[String] = List(red, blue, green)

scala> colors.head
res0: String = red

scala> colors.tail
res1: List[String] = List(blue, green)
```

For a refresher on lists, see Step 8 in Chapter 3. You can find details on using lists in Chapter 16. Lists will also be discussed in Chapter 22, which provides insight into how lists are implemented in Scala.

Arrays

Arrays allow you to hold a sequence of elements and efficiently access an element at an arbitrary position, either to get or update the element, with a zero-based index. Here's how you create an array whose size you know, but for which you don't yet know the element values:

```
scala> val fiveInts = new Array[Int](5)
fiveInts: Array[Int] = Array(0, 0, 0, 0, 0)
```

Here's how you initialize an array when you do know the element values:

```
scala> val fiveToOne = Array(5, 4, 3, 2, 1)
fiveToOne: Array[Int] = Array(5, 4, 3, 2, 1)
```

As mentioned previously, arrays are accessed in Scala by placing an index in parentheses, not square brackets as in Java. Here's an example of both accessing and updating an array element:

```
scala> fiveInts(0) = fiveToOne(4)

scala> fiveInts
res3: Array[Int] = Array(1, 0, 0, 0, 0)
```

Scala arrays are represented in the same way as Java arrays. So, you can seamlessly use existing Java methods that return arrays.[1]

You have seen arrays in action many times in previous chapters. The basics are in Step 7 inChapter 3. Several examples of iterating through the elements of an array with a forexpression are shown in Section 7.3. Arrays also figure prominently in the two-dimensional layout library of Chapter 10.

List buffers

Class List provides fast access to the head of the list, but not the end. Thus, when you need to build a list by appending to the end, consider building the list backwards by prepending elements to the front. Then when you're done, call reverse to get the elements in the order you need.

Another alternative, which avoids the reverse operation, is to use a ListBuffer. A ListBuffer is a mutable object (contained in package scala.collection.mutable), which can help you build lists more efficiently when you need to append. ListBuffer provides constant time append and prepend operations. You append elements with the += operator, and prepend them with the+=: operator. When you're done building, you can obtain a List by invoking toList on theListBuffer. Here's an example:

```
scala> import scala.collection.mutable.ListBuffer
import scala.collection.mutable.ListBuffer

scala> val buf = new ListBuffer[Int]
buf: scala.collection.mutable.ListBuffer[Int] = ListBuffer()

scala> buf += 1
res4: buf.type = ListBuffer(1)

scala> buf += 2
res5: buf.type = ListBuffer(1, 2)

scala> buf
res6: scala.collection.mutable.ListBuffer[Int] =
    ListBuffer(1, 2)

scala> 3 +=: buf
res7: buf.type = ListBuffer(3, 1, 2)

scala> buf.toList
res8: List[Int] = List(3, 1, 2)
```

Another reason to use ListBuffer instead of List is to prevent the potential for stack overflow.If you can build a list in the desired order by prepending, but the recursive algorithm that would be required is not tail recursive, you can use a for expression or while loop and aListBuffer instead. You'll see ListBuffer being used in this way in Section 22.2.

Array buffers

An ArrayBuffer is like an array, except that you can additionally add and remove elements from the beginning and end of the sequence. All Array operations are available, though they are a little slower due to a layer of wrapping in the implementation. The new addition and removal operations are constant time on average, but occasionally require linear time due to the implementation needing to allocate a new array to hold the buffer's contents.

To use an ArrayBuffer, you must first import it from the mutable collections package:

```
scala> import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.ArrayBuffer
```

When you create an ArrayBuffer, you must specify a type parameter, but you don't need to specify a length. The ArrayBuffer will adjust the allocated space automatically as needed:

```
scala> val buf = new ArrayBuffer[Int]()
buf: scala.collection.mutable.ArrayBuffer[Int] =
ArrayBuffer()
```

You can append to an ArrayBuffer using the += method:

```
scala> buf += 12
res9: buf.type = ArrayBuffer(12)

scala> buf += 15
res10: buf.type = ArrayBuffer(12, 15)

scala> buf
res11: scala.collection.mutable.ArrayBuffer[Int] =
    ArrayBuffer(12, 15)
```

All the normal array methods are available. For example, you can ask an ArrayBuffer its length or you can retrieve an element by its index:

```
scala> buf.length
res12: Int = 2

scala> buf(0)
res13: Int = 12
```

Strings (via StringOps)

One other sequence to be aware of is StringOps, which implements many sequence methods. Because Predef has an implicit conversion from String to StringOps, you can treat any string like a sequence. Here's an example:

```
scala> def hasUpperCase(s: String) = s.exists(_.isUpper)
hasUpperCase: (s: String)Boolean

scala> hasUpperCase("Robert Frost")
res14: Boolean = true

scala> hasUpperCase("e e cummings")
res15: Boolean = false
```

In this example, the exists method is invoked on the string named s in the hasUpperCase method body. Because no method named "exists" is declared in class String itself, the Scala compiler will implicitly convert s to StringOps, which has the method. The exists method treats the string as a sequence of characters, and will return true if any of the characters are upper case.[2]

17.2 SETS AND MAPS

You have already seen the basics of sets and maps in previous chapters, starting with Step 10in Chapter 3. In this section, we'll offer more insight into their use and show you a few more examples.

As mentioned previously, the Scala collections library offers both mutable and immutable versions of sets and maps. The hierarchy for sets is shown in Figure 3.2 here, and the hierarchy for maps is shown in Figure 3.3 here. As these diagrams show, the simple namesSet and Map are used by three traits each, residing in different packages.

By default when you write "Set" or "Map" you get an immutable object. If you want the mutable variant, you need to do an explicit import. Scala gives you easier access to the immutable variants, as a gentle encouragement to prefer them over their mutable counterparts. The easy access is provided via the Predef object, which is implicitly imported into every Scala source file. Listing 17.1 shows the relevant definitions:

```
object Predef {
  type Map[A, +B] = collection.immutable.Map[A, B]
  type Set[A] = collection.immutable.Set[A]
  val Map = collection.immutable.Map
  val Set = collection.immutable.Set
  // ...
}
```

Listing 17.1 - Default map and set definitions in Predef.

The "type" keyword is used in Predef to define Set and Map as aliases for the longer fully qualified names of the immutable set and map traits.[3] The vals named Set and Map are initialized to refer to the singleton objects for the immutable Set and Map. So Map is the same asPredef.Map, which is defined to be the same as scala.collection.immutable.Map. This holds both for the Map type and Map object.

If you want to use both mutable and immutable sets or maps in the same source file, one approach is to import the name of the package that contains the mutable variants:

```
scala> import scala.collection.mutable
import scala.collection.mutable
```

You can continue to refer to the immutable set as Set, as before, but can now refer to the mutable set as mutable.Set. Here's an example:

```
scala> val mutaSet = mutable.Set(1, 2, 3)
mutaSet: scala.collection.mutable.Set[Int] = Set(1, 2, 3)
```

Using sets

The key characteristic of sets is that they will ensure that at most one of each object, as determined by ==, will be contained in the set at any one time. As an example, we'll use a set to count the number of different words in a string.

The split method on String can separate a string into words, if you specify spaces and punctuation as word separators. The regular expression "[!,.]+" will suffice: It indicates the string should be split at each place that one or more space and/or punctuation characters exist.

```
scala> val text = "See Spot run. Run, Spot. Run!"
text: String = See Spot run. Run, Spot. Run!

scala> val wordsArray = text.split("[ !,.]+")
wordsArray: Array[String]
  = Array(See, Spot, run, Run, Spot, Run)
```

To count the distinct words, you can convert them to the same case and then add them to a set. Because sets exclude duplicates, each distinct word will appear exactly one time in the set.

First, you can create an empty set using the empty method provided on the Set companion objects:

```
scala>  val words = mutable.Set.empty[String]
words: scala.collection.mutable.Set[String] = Set()
```

Then, just iterate through the words with a for expression, convert each word to lower case, and add it to the mutable set with the += operator:

```
scala> for (word <- wordsArray)
         words += word.toLowerCase

scala> words
res17: scala.collection.mutable.Set[String] =
    Set(see, run, spot)
```

Thus, the text contained exactly three distinct words: spot, run, and see. The most commonly used methods on both mutable and immutable sets are shown in Table 17.1.

Common operations for sets

What it is	What it does
val nums = Set(1, 2, 3)	Creates an immutable set (nums.toString returnsSet(1, 2, 3))
nums + 5	Adds an element (returns Set(1, 2, 3, 5))
nums - 3	Removes an element (returns Set(1, 2))
nums ++ List(5, 6)	Adds multiple elements (returns Set(1, 2, 3, 5, 6))

nums -- List(1, 2)	Removes multiple elements (returns Set(3))
nums & Set(1, 3, 5, 7)	Takes the intersection of two sets (returns Set(1, 3))
nums.size	Returns the size of the set (returns 3)
nums.contains(3)	Checks for inclusion (returns true)
import scala.collection.mutable	Makes the mutable collections easy to access
val words = mutable.Set.empty[String]	Creates an empty, mutable set (words.toString returnsSet())
words += "the"	Adds an element (words.toString returns Set(the))
words -= "the"	Removes an element, if it exists (words.toStringreturns Set())
words ++= List("do", "re", "mi")	Adds multiple elements (words.toString returnsSet(do, re, mi))
words --= List("do", "re")	Removes multiple elements (words.toString returnsSet(mi))
words.clear	Removes all elements (words.toString returns Set())

Using maps

Maps let you associate a value with each element of a set. Using a map looks similar to using an array, except instead of indexing with integers counting from 0, you can use any kind of key. If you import the mutable package name, you can create an empty mutable map like this:

```
scala> val map = mutable.Map.empty[String, Int]
map: scala.collection.mutable.Map[String,Int] = Map()
```

Note that when you create a map, you must specify two types. The first type is for the *keys* of the map, the second for the *values*. In this case, the keys are strings and the values are integers. Setting entries in a map looks similar to setting entries in an array:

```
scala> map("hello") = 1

scala> map("there") = 2

scala> map
res20: scala.collection.mutable.Map[String,Int] =
    Map(hello -> 1, there -> 2)
```

Likewise, reading a map is similar to reading an array:

```
scala> map("hello")
res21: Int = 1
```

Putting it all together, here is a method that counts the number of times each word occurs in a string:

```
scala> def countWords(text: String) = {
         val counts = mutable.Map.empty[String, Int]
         for (rawWord <- text.split("[ ,!.]+")) {
           val word = rawWord.toLowerCase
           val oldCount =
             if (counts.contains(word)) counts(word)
             else 0
           counts += (word -> (oldCount + 1))
         }
         counts
       }
```

```
countWords: (text:
String)scala.collection.mutable.Map[String,Int]

scala> countWords("See Spot run! Run, Spot. Run!")
res22: scala.collection.mutable.Map[String,Int] =
    Map(spot -> 2, see -> 1, run -> 3)
```

Given these counts, you can see that this text talks a lot about running, but not so much about seeing.

The way this code works is that a mutable map, named counts, maps each word to the number of times it occurs in the text. For each word in the text, the word's old count is looked up, that count is incremented by one, and the new count is saved back into counts. Note the use ofcontains to check whether a word has been seen yet or not. If counts.contains(word) is not true, then the word has not yet been seen and zero is used for the count.

Many of the most commonly used methods on both mutable and immutable maps are shown in Table 17.2.

Common operations for maps

What it is	What it does
val nums = Map("i" -> 1, "ii" -> 2)	Creates an immutable map (nums.toString returnsMap(i -> 1, ii -> 2))
nums + ("vi" -> 6)	Adds an entry (returns Map(i -> 1, ii -> 2, vi -> 6))
nums - "ii"	Removes an entry (returns Map(i -> 1))
nums ++ List("iii" -> 3, "v" -> 5)	Adds multiple entries (returns Map(i -> 1, ii -> 2, iii -> 3, v -> 5))
nums -- List("i", "ii")	Removes multiple entries (returns Map())
nums.size	Returns the size of the map (returns 2)
nums.contains("ii")	Checks for inclusion (returns true)
nums("ii")	Retrieves the value at a specified key (returns 2)
nums.keys	Returns the keys (returns an Iterable over the strings"i" and "ii")
nums.keySet	Returns the keys as a set (returns Set(i, ii))
nums.values	Returns the values (returns an Iterable over the integers 1 and 2)
nums.isEmpty	Indicates whether the map is empty (returns false)
import scala.collection.mutable	Makes the mutable collections easy to access
val words = mutable.Map.empty[String, Int]	Creates an empty, mutable map
words += ("one" -> 1)	Adds a map entry from "one" to 1 (words.toStringreturns Map(one -> 1))
words -= "one"	Removes a map entry, if it exists (words.toStringreturns Map())
words ++= List("one" -> 1, "two" -> 2, "three" -> 3)	Adds multiple map entries (words.toString returnsMap(one -> 1, two -> 2, three -> 3))
words --= List("one", "two")	Removes multiple objects (words.toString returnsMap(three -> 3))

Default sets and maps

For most uses, the implementations of mutable and immutable sets and maps provided bythe Set(), scala.collection.mutable.Map(), *etc.*, factories will likely be sufficient. The implementations provided by these factories use a fast lookup algorithm, usually involving a hash table, so they can quickly decide whether or not an object is in the collection.

The scala.collection.mutable.Set() factory method, for example, returns ascala.collection.mutable.HashSet, which uses a hash table internally. Similarly, thescala.collection.mutable.Map() factory returns a scala.collection.mutable.HashMap.

The story for immutable sets and maps is a bit more involved. The class returned by thescala.collection.immutable.Set() factory method, for example, depends on how many elements you pass to it, as shown in Table 17.3. For sets with fewer than five elements, a special class devoted exclusively to sets of each particular size is used to maximize performance. Once you request a set that has five or more elements in it, however, the factory method will return an implementation that uses hash tries.

Similarly, the scala.collection.immutable.Map() factory method will return a different class depending on how many key-value pairs you pass to it, as shown in Table 17.4. As with sets, for immutable maps with fewer than five elements, a special class devoted exclusively to maps of each particular size is used to maximize performance. Once a map has five or more key-value pairs in it, however, an immutable HashMap is used.

The default immutable implementation classes shown in Tables 17.3 and 17.4 work together to give you maximum performance. For example, if you add an element to an EmptySet, it will return a Set1. If you add an element to that Set1, it will return a Set2. If you then remove an element from the Set2, you'll get another Set1.

Table 17.3 - Default immutable set implementations

Number of elements	Implementation
0	scala.collection.immutable.EmptySet
1	scala.collection.immutable.Set1
2	scala.collection.immutable.Set2
3	scala.collection.immutable.Set3
4	scala.collection.immutable.Set4
5 or more	scala.collection.immutable.HashSet

Table 17.4 - Default immutable map implementations

Number of elements	Implementation
0	scala.collection.immutable.EmptyMap
1	scala.collection.immutable.Map1
2	scala.collection.immutable.Map2
3	scala.collection.immutable.Map3
4	scala.collection.immutable.Map4

5 or more	scala.collection.immutable.HashMap

Sorted sets and maps

On occasion you may need a set or map whose iterator returns elements in a particular order. For this purpose, the Scala collections library provides traits SortedSet and SortedMap. These traits are implemented by classes TreeSet and TreeMap, which use a red-black tree to keep elements (in the case of TreeSet) or keys (in the case of TreeMap) in order. The order is determined by the Ordered trait, which the element type of the set, or key type of the map, must either mix in or be implicitly convertible to. These classes only come in immutable variants. Here are some TreeSet examples:

```
scala> import scala.collection.immutable.TreeSet
import scala.collection.immutable.TreeSet

scala> val ts = TreeSet(9, 3, 1, 8, 0, 2, 7, 4, 6, 5)
ts: scala.collection.immutable.TreeSet[Int] =
    TreeSet(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> val cs = TreeSet('f', 'u', 'n')
cs: scala.collection.immutable.TreeSet[Char] =
    TreeSet(f, n, u)
```

And here are a few TreeMap examples:

```
scala> import scala.collection.immutable.TreeMap
import scala.collection.immutable.TreeMap

scala> var tm = TreeMap(3 -> 'x', 1 -> 'x', 4 -> 'x')
tm: scala.collection.immutable.TreeMap[Int,Char] =
    Map(1 -> x, 3 -> x, 4 -> x)

scala> tm += (2 -> 'x')

scala> tm
res30: scala.collection.immutable.TreeMap[Int,Char] =
    Map(1 -> x, 2 -> x, 3 -> x, 4 -> x)
```

17.3 SELECTING MUTABLE VERSUS IMMUTABLE COLLECTIONS

For some problems, mutable collections work better, while for others, immutable collections work better. When in doubt, it is better to start with an immutable collection and change it later, if you need to, because immutable collections can be easier to reason about than mutable ones.

Also, it can be worthwhile to go the opposite way sometimes. If you find some code that uses mutable collections becoming complicated and hard to reason about, consider whether it would help to change some of the collections to immutable alternatives. In particular, if you find yourself worrying about making copies of mutable collections in just the right places, or thinking a lot about who "owns" or "contains" a mutable collection, consider switching some of the collections to their immutable counterparts.

Besides being potentially easier to reason about, immutable collections can usually be stored more compactly than mutable ones if the number of elements stored in the collection is small. For instance an

empty mutable map in its default representation of HashMap takes up about 80 bytes, and about 16 more are added for each entry that's added to it. An empty immutable Map is a single object that's shared between all references, so referring to it essentially costs just a single pointer field.

What's more, the Scala collections library currently stores immutable maps and sets with up to four entries in a single object, which typically takes up between 16 and 40 bytes, depending on the number of entries stored in the collection.[4] So for small maps and sets, the immutable versions are much more compact than the mutable ones. Given that many collections are small, switching them to be immutable can bring important space savings and performance advantages.

To make it easier to switch from immutable to mutable collections, and vice versa, Scala provides some syntactic sugar. Even though immutable sets and maps do not support a true+= method, Scala gives a useful alternate interpretation to +=. Whenever you write a += b, and adoes not support a method named +=, Scala will try interpreting it as a = a + b.

For example, immutable sets do not support a += operator:

```
scala> val people = Set("Nancy", "Jane")
people: scala.collection.immutable.Set[String] =
    Set(Nancy, Jane)

scala> people += "Bob"
<console>:14: error: value += is not a member of
scala.collection.immutable.Set[String]
              people += "Bob"
                     ^
```

However, if you declare people as a var, instead of a val, then the collection can be "updated" with a += operation, even though it is immutable. First, a new collection will be created, and then people will be reassigned to refer to the new collection:

```
scala> var people = Set("Nancy", "Jane")
people: scala.collection.immutable.Set[String] =
    Set(Nancy, Jane)

scala> people += "Bob"

scala> people
res34: scala.collection.immutable.Set[String] =
    Set(Nancy, Jane, Bob)
```

After this series of statements, the people variable refers to a new immutable set, which contains the added string, "Bob". The same idea applies to any method ending in =, not just the+= method. Here's the same syntax used with the -= operator, which removes an element from a set, and the ++= operator, which adds a collection of elements to a set:

```
scala> people -= "Jane"

scala> people ++= List("Tom", "Harry")

scala> people
res37: scala.collection.immutable.Set[String] =
```

```
Set(Nancy, Bob, Tom, Harry)
```

To see how this is useful, consider again the following Map example from Section 1.1:

```
var capital = Map("US" -> "Washington", "France" -> "Paris")
capital += ("Japan" -> "Tokyo")
println(capital("France"))
```

This code uses immutable collections. If you want to try using mutable collections instead, all that is necessary is to import the mutable version of Map, thus overriding the default import of the immutable Map:

```
import scala.collection.mutable.Map  // only change needed!
var capital = Map("US" -> "Washington", "France" -> "Paris")
capital += ("Japan" -> "Tokyo")
println(capital("France"))
```

Not all examples are quite that easy to convert, but the special treatment of methods ending in an equals sign will often reduce the amount of code that needs changing.

By the way, this syntactic treatment works on any kind of value, not just collections. For example, here it is being used on floating-point numbers:

```
scala> var roughlyPi = 3.0
roughlyPi: Double = 3.0

scala> roughlyPi += 0.1

scala> roughlyPi += 0.04

scala> roughlyPi
res40: Double = 3.14
```

The effect of this expansion is similar to Java's assignment operators (+=, -=, *=, *etc.*), but it is more general because every operator ending in = can be converted.

17.4 INITIALIZING COLLECTIONS

As you've seen previously, the most common way to create and initialize a collection is to pass the initial elements to a factory method on the companion object of your chosen collection. You just place the elements in parentheses after the companion object name, and the Scala compiler will transform that to an invocation of an apply method on that companion object:

```
scala> List(1, 2, 3)
res41: List[Int] = List(1, 2, 3)

scala> Set('a', 'b', 'c')
res42: scala.collection.immutable.Set[Char] = Set(a, b, c)

scala> import scala.collection.mutable
import scala.collection.mutable


scala> mutable.Map("hi" -> 2, "there" -> 5)
```

```
res43: scala.collection.mutable.Map[String,Int] =
    Map(hi -> 2, there -> 5)

scala> Array(1.0, 2.0, 3.0)
res44: Array[Double] = Array(1.0, 2.0, 3.0)
```

Although most often you can let the Scala compiler infer the element type of a collection from the elements passed to its factory method, sometimes you may want to create a collection but specify a different type from the one the compiler would choose. This is especially an issue with mutable collections. Here's an example:

```
scala> import scala.collection.mutable
import scala.collection.mutable

scala> val stuff = mutable.Set(42)
stuff: scala.collection.mutable.Set[Int] = Set(42)

scala> stuff += "abracadabra"
<console>:16: error: type mismatch;
 found   : String("abracadabra")
 required: Int
              stuff += "abracadabra"
                       ^
```

The problem here is that stuff was given an element type of Int. If you want it to have an element type of Any, you need to say so explicitly by putting the element type in square brackets, like this:

```
scala> val stuff = mutable.Set[Any](42)
stuff: scala.collection.mutable.Set[Any] = Set(42)
```

Another special situation is if you want to initialize a collection with another collection. For example, imagine you have a list, but you want a TreeSet containing the elements in the list. Here's the list:

```
scala> val colors = List("blue", "yellow", "red", "green")
colors: List[String] = List(blue, yellow, red, green)
```

You cannot pass the colors list to the factory method for TreeSet:

```
scala> import scala.collection.immutable.TreeSet
import scala.collection.immutable.TreeSet

scala> val treeSet = TreeSet(colors)
<console>:16: error: No implicit Ordering defined for
List[String].
       val treeSet = TreeSet(colors)
                             ^
```

Instead, you'll need to create an empty TreeSet[String] and add to it the elements of the list with the TreeSet's ++ operator:

```
scala> val treeSet = TreeSet[String]() ++ colors
treeSet: scala.collection.immutable.TreeSet[String] =
    TreeSet(blue, green, red, yellow)
```

Converting to array or list

If you need to initialize a list or array with another collection, on the other hand, it is quite straightforward. As you've seen previously, to initialize a new list with another collection, simply invoke toList on that collection:

```
scala> treeSet.toList
res50: List[String] = List(blue, green, red, yellow)
```

Or, if you need an array, invoke toArray:

```
scala> treeSet.toArray
res51: Array[String] = Array(blue, green, red, yellow)
```

Note that although the original colors list was not sorted, the elements in the list produced by invoking toList on the TreeSet are in alphabetical order. When you invoke toList or toArray on a collection, the order of the elements in the resulting list or array will be the same as the order of elements produced by an iterator obtained by invoking elements on that collection. Because a TreeSet[String]'s iterator will produce strings in alphabetical order, those strings will appear in alphabetical order in the list resulting from invoking toList on that TreeSet.

Keep in mind, however, that conversion to lists or arrays usually requires copying all of the elements of the collection, and thus may be slow for large collections. Sometimes you need to do it, though, due to an existing API. Further, many collections only have a few elements anyway, in which case there is only a small speed penalty.

Converting between mutable and immutable sets and maps

Another situation that arises occasionally is the need to convert a mutable set or map to an immutable one, or *vice versa*. To accomplish this, you can use the technique shown on the previous page to initialize a TreeSet with the elements of a list. Create a collection of the new type using the empty method and then add the new elements using either ++ or ++=, whichever is appropriate for the target collection type. Here's how you'd convert the immutable TreeSetfrom the previous example to a mutable set, and back again to an immutable one:

```
scala> import scala.collection.mutable
import scala.collection.mutable

scala> treeSet
res52: scala.collection.immutable.TreeSet[String] =
    TreeSet(blue, green, red, yellow)

scala> val mutaSet = mutable.Set.empty ++= treeSet
mutaSet: scala.collection.mutable.Set[String] =
    Set(red, blue, green, yellow)

scala> val immutaSet = Set.empty ++ mutaSet
immutaSet: scala.collection.immutable.Set[String] =
    Set(red, blue, green, yellow)
```

You can use the same technique to convert between mutable and immutable maps:

```
scala> val muta = mutable.Map("i" -> 1, "ii" -> 2)
muta: scala.collection.mutable.Map[String,Int] =
    Map(ii -> 2,i -> 1)

scala> val immu = Map.empty ++ muta
immu: scala.collection.immutable.Map[String,Int] =
    Map(ii -> 2, i -> 1)
```

17.5 TUPLES

As described in Step 9 in Chapter 3, a tuple combines a fixed number of items together so that they can be passed around as a whole. Unlike an array or list, a tuple can hold objects with different types. Here is an example of a tuple holding an integer, a string, and the console:

```
(1, "hello", Console)
```

Tuples save you the tedium of defining simplistic data-heavy classes. Even though defining a class is already easy, it does require a certain minimum effort, which sometimes serves no purpose. Tuples save you the effort of choosing a name for the class, choosing a scope to define the class in, and choosing names for the members of the class. If your class simply holds an integer and a string, there is no clarity added by defining a class namedAnIntegerAndAString.

Because tuples can combine objects of different types, tuples do not inherit from Traversable. If you find yourself wanting to group exactly one integer and exactly one string, then you want a tuple, not a List or Array.

A common application of tuples is returning multiple values from a method. For example, here is a method that finds the longest word in a collection and also returns its index:

```
def longestWord(words: Array[String]) = {
  var word = words(0)
  var idx = 0
  for (i <- 1 until words.length)
    if (words(i).length > word.length) {
      word = words(i)
      idx = i
    }
  (word, idx)
}
```

Here is an example use of the method:

```
scala> val longest =
         longestWord("The quick brown fox".split(" "))
longest: (String, Int) = (quick,1)
```

The longestWord function here computes two items: word, the longest word in the array, and idx, the index of that word. To keep things simple, the function assumes there is at least one word in the list, and it breaks ties by choosing the word that comes earlier in the list. Once the function has chosen which word and index to return, it returns both of them together using the tuple syntax (word, idx).

To access elements of a tuple, you can use method _1 to access the first element, _2 to access the second, and so on:

```
scala> longest._1
res53: String = quick

scala> longest._2
res54: Int = 1
```

Additionally, you can assign each element of the tuple to its own variable,[5] like this:

```
scala> val (word, idx) = longest
word: String = quick
idx: Int = 1

scala> word
res55: String = quick
```

By the way, if you leave off the parentheses you get a different result:

```
scala> val word, idx = longest
word: (String, Int) = (quick,1)
idx: (String, Int) = (quick,1)
```

This syntax gives *multiple definitions* of the same expression. Each variable is initialized with its own evaluation of the expression on the right-hand side. That the expression evaluates to a tuple in this case does not matter. Both variables are initialized to the tuple in its entirety. SeeChapter 18 for some examples where multiple definitions are convenient.

As a note of warning, tuples are almost too easy to use. Tuples are great when you combine data that has no meaning beyond "an A and a B." However, whenever the combination has some meaning, or you want to add some methods to the combination, it is better to go ahead and create a class. For example, do not use a 3-tuple for the combination of a month, a day, and a year. Make a Date class. It makes your intentions explicit, which both clears up the code for human readers and gives the compiler and language opportunities to help you catch mistakes.

17.6 CONCLUSION

This chapter has given an overview of the Scala collections library and the most important classes and traits in it. With this foundation you should be able to work effectively with Scala collections, and know where to look in Scaladoc when you need more information. For more detailed information about Scala collections, look ahead to Chapter 24 and Chapter 25. For now, in the next chapter, we'll turn our attention from the Scala library back to the language and discuss Scala's support for mutable objects.

Footnotes for Chapter 17:

[1] The difference in variance of Scala's and Java's arrays—*i.e.*, whether Array[String] is a subtype of Array[AnyRef]—will be discussed in Section 19.3.

[2] The code given here of Chapter 1 presents a similar example.

[3] The type keyword will be explained in more detail in Section 20.6.

[4] The "single object" is an instance of Set1 through Set4, or Map1 through Map4, as shown inTables 17.3 and 17.4.

[5] This syntax is actually a special case of *pattern matching*, as described in detail in Section 15.7.