Collections

«Previous Next»

This lesson covers:

Basic Data Structures

Scala provides some nice collections.

See Also Effective Scala has opinions about how to use collections.

Lists

```
scala> val numbers = List(1, 2, 3, 4)
numbers: List[Int] = List(1, 2, 3, 4)
```

Sets

Sets have no duplicates

```
scala> Set(1, 1, 2)
res0: scala.collection.immutable.Set[Int] = Set(1, 2)
```

Tuple

A tuple groups together simple logical collections of items without using a class.

```
scala> val hostPort = ("localhost", 80)
hostPort: (String, Int) = (localhost, 80)
```

Unlike case classes, they don't have named accessors, instead they have accessors that are named by their position and is 1-based rather than 0-based.

```
scala> hostPort._1
res0: String = localhost

scala> hostPort._2
res1: Int = 80
```

Tuples fit with pattern matching nicely.

```
hostPort match {
  case ("localhost", port) => ...
```

```
  case (host, port) => ...
}
```

Tuple has some special sauce for simply making Tuples of 2 values: `->`

```
scala> 1 -> 2
res0: (Int, Int) = (1,2)
```

See Also Effective Scala has opinions about destructuring bindings (“unpacking” a tuple).

Maps

It can hold basic datatypes.

```
Map(1 -> 2)
Map("foo" -> "bar")
```

This looks like special syntax but remember back to our discussion of Tuple that `->` can be use to create Tuples.

Map() also uses that variable argument syntax we learned back in Lesson #1: `Map(1 -> "one", 2 -> "two")` which expands into `Map((1, "one"), (2, "two"))` with the first element being the key and the second being the value of the Map.

Maps can themselves contain Maps or even functions as values.

```
Map(1 -> Map("foo" -> "bar"))
```

```
Map("timesTwo" -> { timesTwo(_) })
```

Option

`Option` is a container that may or may not hold something.

The basic interface for Option looks like:

```
trait Option[T] {
  def isDefined: Boolean
  def get: T
  def getOrElse(t: T): T
}
```

Option itself is generic and has two subclasses: `Some[T]` or `None`

Let’s look at an example of how Option is used:

`Map.get` uses `Option` for its return type. Option tells you that the method might not return what you’re asking for.

```
scala> val numbers = Map("one" -> 1, "two" -> 2)
numbers: scala.collection.immutable.Map[java.lang.String,Int] = Map(one -> 1, two -> 2)

scala> numbers.get("two")
res0: Option[Int] = Some(2)

scala> numbers.get("three")
res1: Option[Int] = None
```

Now our data appears trapped in this `Option`. How do we work with it?

A first instinct might be to do something conditionally based on the `isDefined` method.

```
// We want to multiply the number by two, otherwise return 0.
val result = if (res1.isDefined) {
  res1.get * 2
} else {
  0
```

```
}
```

We would suggest that you use either `getOrElse` or pattern matching to work with this result.

`getOrElse` lets you easily define a default value.

```
val result = res1.getOrElse(0) * 2
```

Pattern matching fits naturally with `Option`.

```
val result = res1 match {
  case Some(n) => n * 2
  case None => 0
}
```

See Also Effective Scala has opinions about Options.

Functional Combinators

`List(1, 2, 3) map squared` applies the function `squared` to the elements of the list, returning a new list, perhaps `List(1, 4, 9)`. We call operations like `map` *combinators*. (If you'd like a better definition, you might like Explanation of combinators on Stackoverflow.) Their most common use is on the standard data structures.

map

Evaluates a function over each element in the list, returning a list with the same number of elements.

```
scala> numbers.map((i: Int) => i * 2)
res0: List[Int] = List(2, 4, 6, 8)
```

or pass in a function (the Scala compiler automatically converts our method to a function)

```
scala> def timesTwo(i: Int): Int = i * 2
timesTwo: (i: Int)Int

scala> numbers.map(timesTwo)
res0: List[Int] = List(2, 4, 6, 8)
```

foreach

foreach is like map but returns nothing. foreach is intended for side-effects only.

```
scala> numbers.foreach((i: Int) => i * 2)
```

returns nothing.

You can try to store the return in a value but it'll be of type Unit (i.e. void)

```
scala> val doubled = numbers.foreach((i: Int) => i * 2)
doubled: Unit = ()
```

filter

removes any elements where the function you pass in evaluates to false. Functions that return a Boolean are often called predicate functions.

```
scala> numbers.filter((i: Int) => i % 2 == 0)
res0: List[Int] = List(2, 4)
```

```
scala> def isEven(i: Int): Boolean = i % 2 == 0
isEven: (i: Int)Boolean

scala> numbers.filter(isEven _)
res2: List[Int] = List(2, 4)
```

zip

zip aggregates the contents of two lists into a single list of pairs.

```
scala> List(1, 2, 3).zip(List("a", "b", "c"))
res0: List[(Int, String)] = List((1,a), (2,b), (3,c))
```

partition

`partition` splits a list based on where it falls with respect to a predicate function.

```
scala> val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
scala> numbers.partition(_ % 2 == 0)
res0: (List[Int], List[Int]) = (List(2, 4, 6, 8, 10),List(1, 3, 5, 7, 9))
```

find

find returns the first element of a collection that matches a predicate function.

```
scala> numbers.find((i: Int) => i > 5)
res0: Option[Int] = Some(6)
```

drop & dropWhile

`drop` drops the first i elements

```
scala> numbers.drop(5)
res0: List[Int] = List(6, 7, 8, 9, 10)
```

`dropWhile` removes the first elements that match a predicate function. For example, if we `dropWhile` odd numbers from our list of numbers, `1` gets dropped (but not `3` which is “shielded” by `2`).

```
scala> numbers.dropWhile(_ % 2 != 0)
res0: List[Int] = List(2, 3, 4, 5, 6, 7, 8, 9, 10)
```

foldLeft

```
scala> numbers.foldLeft(0)((m: Int, n: Int) => m + n)
res0: Int = 55
```

0 is the starting value (Remember that numbers is a List[Int]), and m acts as an accumulator.

Seen visually:

```
scala> numbers.foldLeft(0) { (m: Int, n: Int) => println("m: " + m + " n: " + n); m + n }
m: 0 n: 1
```

```
m: 1 n: 2
m: 3 n: 3
m: 6 n: 4
m: 10 n: 5
m: 15 n: 6
m: 21 n: 7
m: 28 n: 8
m: 36 n: 9
m: 45 n: 10
res0: Int = 55
```

foldRight

Is the same as foldLeft except it runs in the opposite direction.

```
scala> numbers.foldRight(0) { (m: Int, n: Int) => println("m: " + m + " n: " + n); m + n }
m: 10 n: 0
m: 9 n: 10
m: 8 n: 19
m: 7 n: 27
m: 6 n: 34
m: 5 n: 40
m: 4 n: 45
m: 3 n: 49
m: 2 n: 52
m: 1 n: 54
res0: Int = 55
```

flatten

flatten collapses one level of nested structure.

```
scala> List(List(1, 2), List(3, 4)).flatten
res0: List[Int] = List(1, 2, 3, 4)
```

flatMap

flatMap is a frequently used combinator that combines mapping and flattening. flatMap takes a function that works on the nested lists and then concatenates the results back together.

```
scala> val nestedNumbers = List(List(1, 2), List(3, 4))
nestedNumbers: List[List[Int]] = List(List(1, 2), List(3, 4))

scala> nestedNumbers.flatMap(x => x.map(_ * 2))
res0: List[Int] = List(2, 4, 6, 8)
```

Think of it as short-hand for mapping and then flattening:

```
scala> nestedNumbers.map((x: List[Int]) => x.map(_ * 2)).flatten
res1: List[Int] = List(2, 4, 6, 8)
```

that example calling map and then flatten is an example of the “combinator”-like nature of these functions.

See Also Effective Scala has opinions about flatMap.

Generalized functional combinators

Now we’ve learned a grab-bag of functions for working with collections.

What we’d like is to be able to write our own functional combinators.

Interestingly, every functional combinator shown above can be written on top of fold. Let’s see some examples.

```
def ourMap(numbers: List[Int], fn: Int => Int): List[Int] = {
  numbers.foldRight(List[Int]()) { (x: Int, xs: List[Int]) =>
    fn(x) :: xs
  }
}

scala> ourMap(numbers, timesTwo(_))
res0: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
```

Why List[Int]()? Scala wasn't smart enough to realize that you wanted an empty list of Ints to accumulate into.

Map?

All of the functional combinators shown work on Maps, too. Maps can be thought of as a list of pairs so the functions you write work on a pair of the keys and values in the Map.

```
scala> val extensions = Map("steve" -> 100, "bob" -> 101, "joe" -> 201)
extensions: scala.collection.immutable.Map[String,Int] = Map((steve,100), (bob,101), (joe,201))
```

Now filter out every entry whose phone extension is lower than 200.

```
scala> extensions.filter((namePhone: (String, Int)) => namePhone._2 < 200)
res0: scala.collection.immutable.Map[String,Int] = Map((steve,100), (bob,101))
```

Because it gives you a tuple, you have to pull out the keys and values with their positional accessors. Yuck!

Lucky us, we can actually use a pattern match to extract the key and value nicely.

```
scala> extensions.filter({case (name, extension) => extension < 200})
res0: scala.collection.immutable.Map[String,Int] = Map((steve,100), (bob,101))
```

Why does this work? Why can you pass in a partial pattern match?

Stay tuned for next week!

Built at @twitter by @stevej, @marius, and @lahosken with much help from @evanm, @sprsquish, @kevino, @zuercher, @timtrueman, @wickman, and @mccv; Russian translation by appigram; Chinese simple translation by jasonqu; Korean translation by enshahar;