Working with Scala Collections

A functional primer

Immutable vs Mutable

- Immutable by default
- Unless you request otherwise
- In **mutable** package, like mutable.LinearSeq

Arrays

- Arrays preserve order
- can contain duplicates
- are mutable

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

scala> numbers(3) = 10
```

Lists

- Lists preserve order, can contain duplicates, and are immutable.

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

Sets

- Sets do not preserve order and have no duplicates

```
scala> val numbers = Set(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
numbers: scala.collection.immutable.Set[Int] = Set(5, 1, 2, 3, 4)
```

Tupple

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

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

scala> hostPort._1
res0: String = localhost

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

Map

- A Map is an Iterable consisting of pairs of keys and values (also named mappings or associations)
- A Map is an Iterable consisting of pairs of keys and values (also named mappings or associations)
- Scala offers an implicit conversion that lets you write key -> value as an alternate syntax for the pair (key, value)

```
Map("x" -> 24, "y" -> 25, "z" -> 26)
// or
Map(("x", 24), ("y", 25), ("z", 26))
```

Lazy

- Lazy is good - maybe the work will not be needed? A Stream is like a list except that its elements are computed lazily
- Because of this, a stream can be infinitely long. Only those elements requested are computed
- Streams have the same performance characteristics as lists.
- The example computes a stream that contains a Fibonacci sequence starting with the given two numbers
- The tricky part is computing this sequence without causing an infinite recursion
  - If the function used :: instead of #::, then every call to the function would result in another call, thus causing an infinite recursion. Since it uses #::, though, the right-hand side is not evaluated until it is requested.

```
scala> def fibFrom(a: Int, b: Int): Stream[Int] = a #:: fibFrom(b,
a + b)
 fibFrom: (a: Int,b: Int)Stream[Int]

scala> val fibs = fibFrom(1, 1).take(7)
fibs: scala.collection.immutable.Stream[Int] = Stream(1, ?)

scala> fibs.toList
res9: List[Int] = List(1, 1, 2, 3, 5, 8, 11)
```

Iterator example

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

scala> list.foreach { price => println(price)}
1
2
3
4

scala> list.foreach { println}
1
2
3
4

scala> list foreach println
1
2
3
4
```

Overview of Standard methods

map
foreach
update
apply
find
exists
filter
forall
zip

map

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

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

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

foreach

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

- it returns nothing.

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

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)
```

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

- **drop** drops the first i elements

```
scala> val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

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

foldLeft / foldRight

- In essence, fold takes data in one format and gives it back to you in another.
- The fold method for a List takes two arguments; the start value and a function. This function also takes two arguments; the accumulated value and the current item in the list.
    - reduce in Java

```
val numbers = List(5, 4, 8, 6, 2)
numbers.fold(0) { (z, i) =>
  a + i
}
// result = 25
```

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)
```

Lab Time!

collections_09