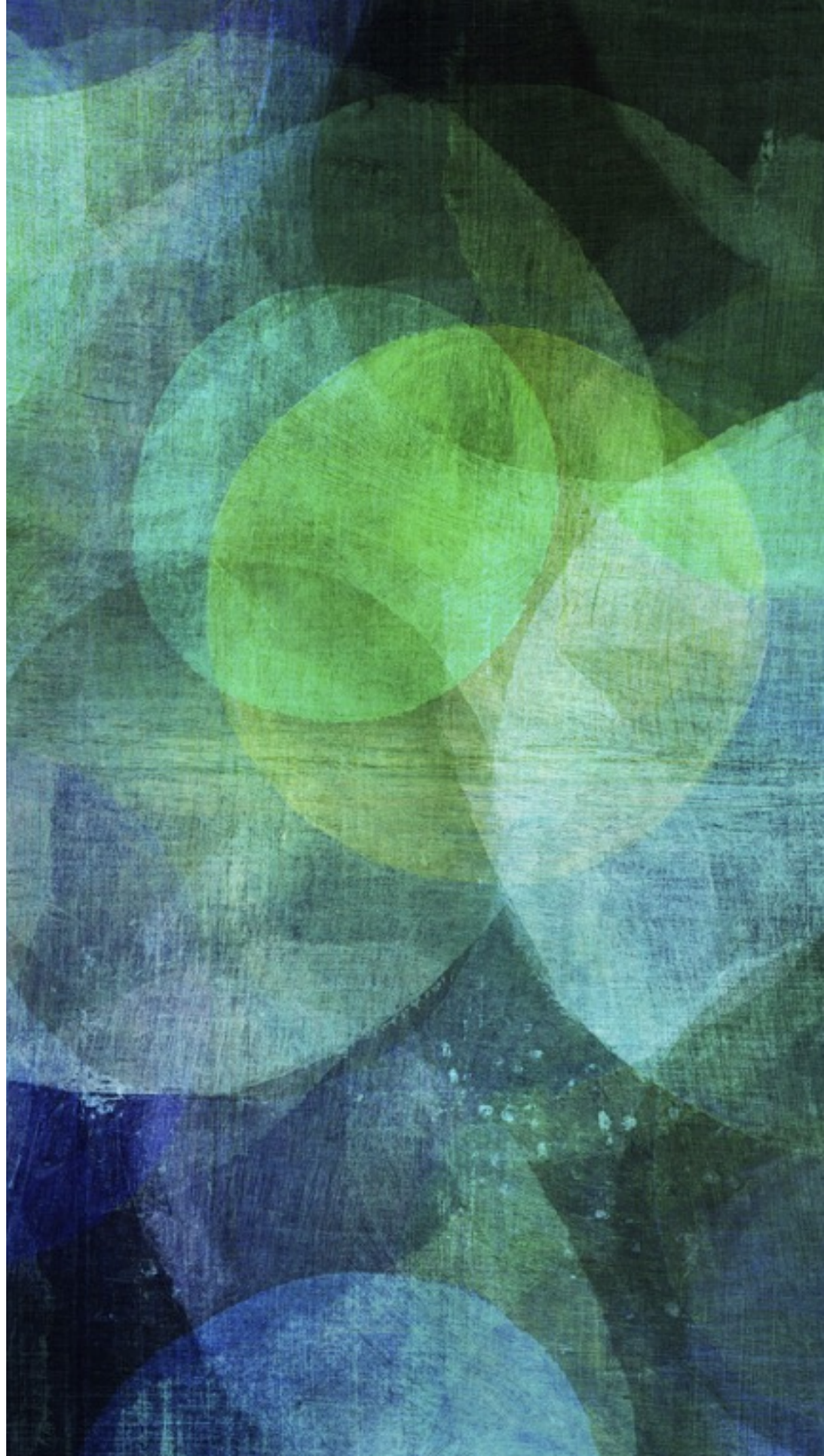
SCALA
PART THREE

Collections



PART 3 OUTLINE

- Basic Data Structures (Arrays, Lists, Sets, etc.)
- Option (Some, None)
- Functional Combinators (map, filter, flatMap, zip, etc.)

BASIC DATA STRUCTURES

ARRAYS

```
val numbers = Array(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
numbers(3)
numbers(3) = 10
numbers(3)
```

- Arrays preserve order, can contain duplicates, and are mutable.

LISTS

```
val numbers = List(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
```

```
numbers(3)
```

```
val n: List[Int] = 1 :: 2 :: 3 :: Nil
```

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

```
numbers(3) = 4
```

SETS

```
val numbers = Set(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
```

- Sets do not preserve order and have no duplicates

TUPLES

```
val hostPort = ("localhost", 80)
```

- A tuple groups together simple logical collections of items without using a class (think database or Excel row)

```
hostPort._1
hostPort._2
```

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

TUPLES AND PATTERN MATCHING

```
hostPort match {
  case ("localhost", port) => println("match")
  case (host, port) => println("oh did we?")
}
```

MAPS

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

- Maps on key/value pairs

MORE MAPS

```
Map(1 -> Map("foo" -> "bar"))
Map("timesTwo" -> { timesTwo(_) })
```

- Maps can themselves contain Maps or even functions as values.

OPTION – SCALA AND NULLS

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

OPTION

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

OPTION USAGE EXAMPLE

- Map uses Option in return types…

```
val numbers = Map("one" -> 1, "two" -> 2)
numbers.get("two")
numbers.get("three")
```

OPTION GETORELSE

```
val result = numbers.getOrElse("three",3) * 3
```

OPTION AND PATTERN MATCHING

```
val result = numbers.get("three") match {
  case Some(n) => n * 2
  case None => 0
}
```

COLLECTION DATA TYPES CONCLUSION

- Any questions?

- Next on to combinators (also found in Spark APIs)

MAP

- `map` evaluates a function over each element in the list, returning a list with the same number of elements (might help to think of one-to-one)

```
val n = List(1, 2, 3, 4)
n.map((i: Int) => i * 2)
```

- or pass in a function

```
def timesTwo(i: Int): Int = i * 2
n.map(timesTwo)
```

FOREACH

```
n.foreach(println)
```

- foreach is like `map` but *returns nothing*
- Try it…

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

FILTER

```
n.filter((i: Int) => i % 2 == 0)
```

- Functions that return a Boolean are often called predicate functions

```
def isEven(i: Int): Boolean = i % 2 == 0
n.filter(isEven)
```

ZIP

```
List(1, 2, 3).zip(List("a", "b", "c"))
```

- Aggregates the contents of two lists into a single list of pairs

PARTITION

```
val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
numbers.partition(_ % 2 == 0)
```

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

FIND

```
numbers.find((i: Int) => i > 5)
```

- Returns the ***first*** element of a collection that matches a predicate function.

DROP

```
numbers.drop(5)
```

- drops the first x elements

FOLDLEFT

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

- // 0 is the starting value (Remember that numbers is a List[Int]), and `m` acts as an accumulator.
- Try with a smaller list and different starting value
- To see visually

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

FLATTEN

```
List(List(1, 2), List(3, 4)).flatten
```

- `flatten` collapses one level of nested structure.

FLATMAP

- often confused with `map`. See `map` vs `flatMap`

- I think it helps to reverse the name to mapFlat

FLATMAP

```
val nestedNumbers = List(List(1, 2), List(3, 4))
nestedNumbers.flatMap(x => x.map(_ * 2))
```

- `flatMap` takes a function that works on the nested lists and then concatenates the results back together.

FLATMAP

- mapFlat....

```
nestedNumbers.map((x: List[Int]) => x.map(_ * 2)).flatten
```

SCALA PART 3 CONCLUSION

- Any questions?

- Exercise: Find all the numbers divisible by 3 in the following:

```
val divByThree = List(List(1,2,3), List(3,4,5), List(6,7,8,9))
```

- BTW- you are now fairly dangerous with Scala.

EXERCISE SOLUTION

```
divByThree.flatMap(i => i.filter{ _ % 3 == 0})
```