The number of datastructure implementations in Scala are humongous

Collections

1. Scala's default collections are immutable.
2. They are based on persistent datastructures, which is an awesome topic in itself
3. What we need to understand is that these datastructures while being immutable are efficient

Sequences

```
val array = Array[Int]()
```

Go to collections

1. Mutable - ArrayBuffer
2. LinkedList based - List()
3. Array based - Vector, which is just like an ArrayList - random access

Transforming collections - the map and the filter

Map

(Show presentation NOW !!!)

```
val list = List(1, 2, 3, 4, 5)
val mapped = list.map(each => each * 2)
```

mkString

```
println (mapped.mkString(","))
```

Filter

(Show presentation NOW !!!)

```
list.filter(each => (each * 2) < 7)
```

Fold

(Show presentation NOW !!!)

```
list.foldLeft(0)((acc, curr) => acc + curr)
```

Reduce

```
list.reduce((acc, curr) => acc + curr)
```

Option

Option = Some + None It's just a collection with a single value or NOT.

Either it has some value or nothing at all

```
Some ("Hello")
None
```

```
val opt = Option ("Hello")
print (opt)
```

Maps

```
val map = Map("a" -> 1, "b" -> 2, "c" -> 3)

map("a")

map.get("a")
```

Mutable maps

```
val mutableMap = mutable.Map[String, Int]()

mutableMap += ("a" -> 1)
mutableMap += ("b" -> 2)


mutableMap.mkString(",")
```

Tuples

Show with Control Shift P

```
val pair = ("a", 5)
val threeTup = (1, 2, "c")
```

Underscore reference of Tuple

```
println(pair._2) //Not recommended

val (one, two) = pair

println(two)
```

Higher order functions

```
def multTwo(int1: Int, int2: Int): Int = {
  int1 * int2
}

multTwo(2, 3)


def binaryOp(int1: Int, int2: Int)(op: (Int, Int) => Int): Int = {
  op(int1, int2)
}

binaryOp(2, 3)((x, y)=> x+y)
binaryOp(2, 3)((x, y)=> x*y)
```

List unzip

```
val list2 = List(pair, pair, pair)
val (alpha, nums) = list2.unzip
```

Switching between Java and Scala collections

```
import scala.collection.JavaConverters._

val javaList: java.util.List[Int] = list.asJava
```

Pattern matching (Not covered yet)

```
def guessTheList(list: List[Int]): Boolean = {

  list match {
    case 1 :: 2 :: _ =>
      println("My kinda list")
      true
    case 3 :: 4 :: _ =>
      println("Nope")
      false
    case _ => println("Yikes !!"); false

  }
}

def guessTheType(any: Any): String = any match {
  case list:List[_] => s"this is a list: ${list.mkString(",")}"
  case set:Set[_] => "this is a set"
  case _ => "stranger things"
}

guessTheList(list)
guessTheList(List(3, 1, 2))


guessTheType(list)
guessTheType(map)
```