Collections & FP Basics

Scala Training, 2016

Contents

Functions

- Recap
- Higher Order Functions & Currying
- Functions as objects
- Values, Literals, Types

Collections

- Basic types, operators
- Immutability with Collections
- Functional primitives on collections

For Comprehensions

Recap on functions

- What's in a function?
  - What's the structure of a function?
  - What's the return value of a function?

Recap on functions

- What's in a function?
  - What's the structure of a function?
  - What's the return value of a function?
- *Stack* vs. *Tail* recursion?

Recap on functions

- What's in a function?
  - What's the structure of a function?
  - What's the return value of a function?
- *Stack* vs. *Tail* recursion?
- Does HOF ring a bell?

Higher Order Functions

- A function taking another function as a parameter is a HOF

```
def fun(f: Int => Int, value: Int): Int =
  f(value)
```

Higher Order Functions

- A function taking another function as a parameter is a HOF

```
def fun(f: Int => Int, value: Int): Int =
  f(value)
```

- A function returning another function as a result is a HOF

```
def morefun: Int => Int = {
  def inc(x: Int) = x + 1
  inc
}
```

Higher Order Functions

- A function taking another function as a parameter is a HOF

```
def fun(f: Int => Int, value: Int): Int =
  f(value)
```

- A function returning another function as a result is a HOF

```
def morefun: Int => Int = {
  def inc(x: Int) = x + 1
  inc
}
```

- HOF's are heavily used in collections

Currying

- functions with multiple parameter lists

```
def curriedf(p1: Int, p2: String)(p3: String) = {}
```

Currying

- functions with multiple parameter lists

```
def curriedf(p1: Int, p2: String)(p3: String) = {}
```

- curried function called with fewer parameter lists?

```
curriedf(2, "hello") // ?
```

Currying

- functions with multiple parameter lists

```
def curriedf(p1: Int, p2: String)(p3: String) = {}
```

- curried function called with fewer parameter lists?

```
curriedf(2, "hello") // ?
```

no problem!

Currying

- functions with multiple parameter lists

```
def curriedf(p1: Int, p2: String)(p3: String) = {}
```

- curried function called with fewer parameter lists?

```
curriedf(2, "hello") // ?
```

no problem!
just a function with the remaining parameter lists

Currying

- functions with multiple parameter lists

```
def curriedf(p1: Int, p2: String)(p3: String) = {}
```

- curried function called with fewer parameter lists?

```
curriedf(2, "hello") // ?
```

no problem!
just a function with the remaining parameter lists

Quiz:

```
def sum(x: Int)(y: Int) = x + y
def increment = sum(1)
```

- What's `increment`'s return type?
- ~~If you designed *Scala*, how would your compiler rewrite a curried function?~~

Functions as Objects

- Scala is functional and runs on the JVM...

Functions as Objects

- Scala is functional and runs on the JVM...
  ...where everything is an object.

Functions as Objects

- Scala is functional and runs on the JVM...
  ...where everything is an object.
  How do you upgrade functions to 1st class?

Functions as Objects

- Scala is functional and runs on the JVM...
  ...where everything is an object.
  How do you upgrade functions to 1st class?
- **All functions in Scala are objects!**

Functions as Objects

- Scala is functional and runs on the JVM...
  ...where everything is an object.
  How do you upgrade functions to 1st class?
- **All functions in Scala are objects!**
- Functions are rewritten as *objects* with an *apply* method
- Make an object "callable" by defining an *apply* method on it

Functions as Objects

- Scala is functional and runs on the JVM...
  ...where everything is an object.
  How do you upgrade functions to 1st class?
- **All functions in Scala are objects!**
- Functions are rewritten as *objects* with an *apply* method
- Make an object "callable" by defining an *apply* method on it

```
object HelloWorld {
  def apply() = println("Hello, world!")
}

// what's the
// a) output?
// b) return value?
HelloWorld()
```

Function values

- Functions can be assigned to values
- This works as functions are objects

```
def increment(x: Int) = x + 1
val inc = increment

// an equivalent way of putting it
object increment extends Function1[Int, Int] {
  apply(value: Int): Int = value + 1
}
// notice the underscore - more on this later
val inc = increment.apply _
```

Function literals, lambdas

- Can be used as any functions
- Can be passed as parameters

```
// e.g. (x: Int) => x % 2 == 0
List(1, 2, 3).filter((x: Int) => x % 2 == 0)
```

- Alternative syntax, more syntactic sugar:

```
List(1, 2, 3).filter(x => x % 2 == 0)
List(1, 2, 3).filter(_ % 2 == 0)
```

Function types

```
trait Function1[-T1, +R] {
  def apply(v1: T1): R
}
```

- The standard library `FunctionN[A1, ... AN, B]` type
- A1 ... AN are the parameter types
- B is the result type
- N goes up to 22
- `Int => Int` is syntax sugar for `Function1[Int, Int]`

Scala Collection Hierarchy

- Traversable is the root of all, Iterable is its subtype
- The 3 large subtypes of Iterable:
  - Set: does not allow duplicates

```
val set = Set(1, 2, 3)
```

  - Map: key-value associations

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

  - Seq: ordered container

```
val seq = Seq(1, 2, 3)
```

Immutability

- The scala.collection package contains two subpackages
  - scala.collection.mutable
  - scala.collection.immutable
- Mutable collections can be updated as a side effect
- Immutable collections by contrast never change
- Immutable collections operations always return new collections
- scala.Predef defines type aliases to immutable collections

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

filter

```
trait Traversable[A] {
  def filter(f: A => Boolean): Traversable[A]
}
```

- Takes a predicate function as a parameter
- Returns a new collection with the items satisfying it

```
List(1, 2, 3, 4).filter(_ % 2 == 0) // List(2, 4)
```

map

```
trait Traversable[A] {
  def map(f: A => B): Traversable[B]
}
```

- Applies the argument function to every element in the collection
- Returns a new collection with the transformed elements

```
// List(1, 0, 1, 0)
List(1, 2, 3, 4).map(_ % 2)

// List(false, true, false, true)
List(1, 2, 3, 4).map(_ % 2 == 0)
```

flatMap

```
trait Traversable[A] {
  def flatMap(f: A => Traversable[B]): Traversable[B]
}
```

- Applies a function to every item in the collection
- Each application returns a new collection, not a single item
- The result is the aggregation of all subcollections

```
// List(1, -1, 2, -2, 3, -3)
List(1, 2, 3).flatMap(x => List(x, -x))
```

For Comprehensions

- they're a pile of syntactic sugar

For Comprehensions

- they're a pile of syntactic sugar
- they are *expressions*...

For Comprehensions

- they're a pile of syntactic sugar
- they are *expressions*...
- ... not loops

For Comprehensions

- they're a pile of syntactic sugar
- they are *expressions*...
- ... not loops
- ... so they evaluate to some value

For Comprehensions

- they're a pile of syntactic sugar
- they are *expressions*...
- ... not loops
- ... so they evaluate to some value
- we say they *yield* a result

For Comprehensions

- they're a pile of syntactic sugar
- they are *expressions*...
- ... not loops
- ... so they evaluate to some value
- we say they *yield* a result

```
val numbers = List(1, 2, 3)
val doubles = for (x <- numbers) yield 2 * x
// List(2, 4, 6)
```

For Comprehensions

- they're a pile of syntactic sugar
- they are *expressions*...
- ... not loops
- ... so they evaluate to some value
- we say they *yield* a result

```
val numbers = List(1, 2, 3)
val doubles = for (x <- numbers) yield 2 * x
// List(2, 4, 6)
```

- `x <- numbers` is a generator
- numbers is the iteratee
- x is a local value bound to the current element

For Comprehensions

- Scala allows for multiple generators

```
val a = List(1, 2)
val b = List(3, 4)

// Yields the cartesian product of a & b
for (x <- a; y <- b) yield (x, y)
// multi-line syntax
for {
  x <- a
  y <- b
} yield (x, y)

// More sugar: human readable generators
for (x <- 1 to 10) yield Math.pow(2, x)
```

For Comprehensions: Filters & Definitions

- For comprehensions allow defining filters

```
for {
  x <- 1 to 4 if x % 2 == 0
} yield 10 * x
```

- Definitions are made using the '='

```
for {
  p    <- persons
  name = p.name
  age  = p.age
} yield s"$name->$age"
```

Melting the sugar

- For comprehensions are translated to standard primitives

Melting the sugar

- For comprehensions are translated to standard primitives

```
for (x <- List(1,2,3)) yield x % 2
List(1,2,3).map(x => x % 2)
```

Melting the sugar

- For comprehensions are translated to standard primitives

```
for (x <- List(1,2,3)) yield x % 2
List(1,2,3).map(x => x % 2)
```

```
for (x <- List(1,2,3,4) if x > 2) yield x + 1
List(1,2,3,4).filter(x => x > 2).map(x => x + 1)
```

Melting the sugar

- For comprehensions are translated to standard primitives

```
for (x <- List(1,2,3)) yield x % 2
List(1,2,3).map(x => x % 2)
```

```
for (x <- List(1,2,3,4) if x > 2) yield x + 1
List(1,2,3,4).filter(x => x > 2).map(x => x + 1)
```

```
for (x <- List(1,2,3); y <- 1 to x) yield x * y
List(1,2,3).flatMap(x => (1 to x).map(y => x * y))
```

Practice!

http://workshop.rosedu.org/2016/sesiuni/scala/lab3