Advanced FP Concepts

Scala training, 2016

Outline

Recap

- Class Params vs. Class Fields
- Singleton Objects
- Named & Default Arguments
- Case Classes
- Traits & Multiple Inheritance
- Collections: Mutable vs. Immutable
- Pattern Matching
- Option / Try

Recursion

- What's a recursive method?
- Recursive methods need to explicitly state their return type
- Let's define the factorial function in Scala
- What issues do we know with classic recursion?

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

Tail Recursion

- The better method recursion
- Doesn't cause stack overflow
- The compiler can optimize the call if you let it know
- Annotate tail recursive methods with *@tailrec*
- The annotation also checks at compile time if the method is indeed tail recursive

```
@tailrec
def factorial(n: Int, acc: Int = 1): Int =
  if (n == 0) acc else factorial(n - 1, n * acc)
```

Local Methods

- You can define local methods just as you can define local variables
- The compiler translates them into private methods
- Use them to structure better your code
- The improved version of factorial:

```
def factorial(n: Int): Int = {
  @tailrec def factorial(n: Int, acc: Int): Int =
    if (n == 0) acc else factorial(n - 1, n * acc)
  factorial(n, 1)
}
```

PartialFunctions

- What's a PF?
- As the name suggest, a function that's defined only for some input values
- That means its domain of values might be restricted
- Do we know (or guess) any PF?

PartialFunctions

- What's a PF?
- As the name suggest, a function that's defined only for some input values
- That means its domain of values might be restricted
- Do we know (or guess) any PF? Map

```
trait PartialFunction[A, B] extends (A => B) {
  def isDefinedAt(x: A): Boolean
}
```

Map

- A Map is defined only for a set of values
- Trying to get an unexisting key value results in an exception

```
trait Map[A, B] extends PartialFunction[A, B] {
 def get(key: A): Option[B]
 def apply(v: A): B
 def isDefinedAt(v: A): Boolean
}
```

Other PFs

- Is a fraction defined for any values?

```
val fraction = new PartialFunction[Int, Int] {
 def apply(d: Int): Int = 42 / d
 def isDefinedAt(d: Int): Boolean = d != 0
}
```

Useful PF operators

- *orElse*
  - Use it to chain multiple partial functions having the same type
  - It's useful to fold multiple domains into a single one
  - Think that you can build a switch or pattern match like structure with them
- *applyOrElse*
  - Use it to call the partial function and use a fallback function

```
trait PFunc[A, B] {
  def orElse(that: PFunc[A, B]): PFunc[A, B]
  def applyOrElse(x: A, default: A => B): B
}
```

A chain sample

```
val handleSuccess: PartialFunction[Try[_], String] = {
 case Success(value) => "Successful handling \$value"
}

val handleFailure: PartialFunction[Try[_], String] = {
 case Failure(t) => "Failed with \${t.getMessage}"
}

val completionHandler =
 handleSuccess orElse handleFailure
```

PF Literals

- A PF literal is given within a block with case alternatives
- Watch out for non-exhaustive matches
- The code can blow up with a *MatchError*
- Use them to enhance code readability

```
case class PhoneBookEntry(name: String, number: String)

// Assume `fetchPhoneBook` is implemented
val phoneBook: Map[Char, List[PhoneBookEntry]] =
    fetchPhoneBook()

// To keep only non empty lists of entries
phoneBook filter {
  case (_, entries) => entries.nonEmpty
}
```

Well-known *HOFs*

- *map* - transform a collection element by element

```
List(1,2,3).map(_.toString)
```

- *flatMap* - transform elements to collections themselves

```
List(1,2,3).flatMap(List(_))
```

- *filter* - filter collections via some predicate

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

foreach, *forall* & *exists*

- *foreach*
  - Run the given piece of code for each element in the collection
  - It's useful for doing side effect ops for each element (i.e. I/O)

```
List(1,2,3).foreach(println)
```

- *forall* - tests that a predicate holds for all elements in a coll.

```
List(1,2,3).forall(_ % 2 == 0)
```

- *exists* - tests that a predicate holds for some element in a coll.

```
List(1,2,3).exists(_ % 2 == 0)
```

Folds

- Use *foldLeft* & *foldRight* to collapse collections in single values
- The given function is to reduce all elements in an accumulator
- The accumulator is initially set to the first parameter list

```
trait Traversable[A] {
  def  foldLeft[B](z: B)(op: (B, A) => B): B
  def foldRight[B](z: B)(op: (A, B) => B): B
}

List(1,2,3).foldLeft(1)(_ * _)
```

groupBy

- creates a map of groups based on the argument function
- the new map values are collections of the same type as the initial collection

```
trait TraversableLike[A, Repr] {
  def groupBy[K](f: A => K): Map[K, Repr]
}

// Map(false -> List(1, 3), true -> List(2))
List(1,2,3).groupBy(_ % 2 == 0)
```

Multiple Parameter Lists

- **currying** is about having multiple parameter lists
- an incomplete call results in a function
- to completely invoke a method, specify *all* params

```
def normalAdd(x: Int, y: Int) = x + y
def curriedAdd(x: Int)(y: Int) = x + y
```

Partially Applied Functions

- You can omit passing arguments
- Multiple argument lists can be replaced with '_'
- The code doesn't compile either

```
def curriedAdd(x: Int)(y: Int) = x + y
def addOne = add(1) _
```

The Magnet Pattern

- Is there a way to lift overloaded methods into functions?
- A teaser on magnets

```
def foo(v: Int): Unit
def foo(v: String): Unit

// What happens?
val f = foo _
```

Lazy Evaluation

- Scala supports lazy evaluation through 'lazy val's
- The expression is not evaluated immediately, but at its first use
- Subsequent calls of that value won't re-evaluate it
- The initialization is guaranteed to happen exactly once, even in a multi-threaded scenario

```scala
// Output for each line
val two = 1 + 1
two

lazy val two = 1 + 1
two
```

Downsides

- Lots of concurrency issues can arise
- lazy vals somehow allow cyclic dependencies
- Sequential initialization due to monitor on instance
- Deadlock on concurrent access of lazy vals without cycle
- Deadlock in combination with other synchronization constructs