Intro To Scala

Ian Murray

July 8, 2014

Scala Gives You `Option[S]`

Overview

- Data Structures
  - defining
  - manipulating
  - collections library
- Control Structures
- Functions
- Handling Failure
- Abstractions
- Handling State
- Handling Effects

Data Structures

Classic OOP

Classic **Object Oriented** encapsulation of data and state transitions *is* possible. . .

Classic OOP

```scala
class ClassicOOPerson(

    /** Constructor Args **/
    private var _firstName: String,
    private var _lastName: String,
    private var _age: Int) {

  /** Getters **/
  def firstName = _firstName
  def lastName  = _lastName
  def age       = _age

  /** Setters **/
  def firstName_=(s: String) = { _firstName = s }
  def lastName_=(s: String)  = { _lastName = s }

  /** Public methods **/
  def fullName = s"${firstName} ${lastName}"
  def celebrateBirthday(): Unit = { _age += 1 }
}
```

Case Classes

Isn't this supposed to be a **Functional** programming talk?

Case Classes

```scala
case class PersonData(
    firstName: String,
    lastName: String,
    age: Int) {

  val fullName = s"${firstName} ${lastName}"
}
```

Defines an immutable data structure:

```
scala> val p = Person("John", "Doe", 43)
p: Person = Person(John,Doe,43)

scala> p.fullName
res0: String = John Doe

scala> p.firstName = "Fred"
<console>:8: error: reassignment to val
       p.firstName = "Fred"
```

Case Classes

The compiler expands the given case class definition with further methods...

Case Classes

Deep Structural Equality

```
scala> p == Person("John", "Doe", 43)
res3: Boolean = true

// reference equality can be checked with ‘eq‘ method.
scala> p eq Person("John", "Doe", 43)
res7: Boolean = false
```

Hash Code

Both of which allow us to safely use instances in collections:

```
scala> val s = Set(p)
s: scala.collection.immutable.Set[Person] = Set(Person(John,Doe,43))

scala> s contains Person("John", "Doe", 43)
res5: Boolean = true
```

Case Classes

String Representation

```
scala> println(p)
Person(John,Doe,43)
```

Copy Method

```
scala> p.copy(firstName = "Jane")
res10: Person = Person(Jane,Doe,43)
```

Case Classes

Companion Object

Compiler will generate a *companion object* associated with the case class, and define two special methods on it:

```
// singleton object generated by the compiler
object Person {
  def apply(firstName: String,
            lastName: String,
            age: Int): Person = new Person(firstName, lastName, age)


  // an extractor, we'll visit this shortly
  def unapply = ???
}
```

In Scala, an `apply(...)` method on a class or object is special:

```
val p1 = Person("John", "Doe", 43)
val p2 = new Person("John", "Doe", 43)
```

Companion Objects

`objects` in general:

- Singletons
- Module-like
- **Can** hold (global) state
- **Cannot** be inherited from.
- **Can** extend from other classes or traits.

Companion Objects

Companion `objects`:

- special visibility privelages between companion object and associated class
  - both can access each other's private members
- used in **implicit resolution**.
- generally used to provide factory-like functions

Case Classes

Manipulating Data

1. Methods defined on the class
2. Functions defined on a module

Methods Defined On The Class

```scala
case class Person(
    firstName: String,
    lastName: String,
    age: Int) {

  val fullName = s"${firstName} ${lastName}"

  def celebrateBirthday(): Person = copy(age=age+1)
  def sameFamilyAs(other: Person) = lastName == other.lastName
}
```

Usage:

```
scala> val p1 = Person("John", "Doe", 43)
p1: Person = Person(John,Doe,43)

scala> val p2 = Person("Jane", "Doe", 43)
p2: Person = Person(Jane,Doe,43)

scala> p1.celebrateBirthday()
res0: Person = Person(John,Doe,44)

scala> p1 sameFamilyAs p2
res1: Boolean = true

scala> p1.sameFamilyAs(p2)
res2: Boolean = true
```

Functions Defined In A Module

```scala
object People {

  case class Person(
      firstName: String,
      lastName: String,
      age: Int) {

    val fullName = s"${firstName} ${lastName}"
  }

  def celebrateBirthdayOf(p: Person): Person = p.copy(age = p.age + 1)
  def fromSameFamily(p1: Person, p2: Person) = p1.lastName == p2.lastName

}
```

Usage:

```scala
scala> import People._
import People._

scala> val p1 = Person("John", "Doe", 43)
p1: People.Person = Person(John,Doe,43)

scala> val p2 = Person("Jane", "Doe", 43)
p2: People.Person = Person(Jane,Doe,43)

scala> celebrateBirthdayOf(p1)
res3: People.Person = Person(John,Doe,44)

scala> fromSameFamily(p1, p2)
res4: Boolean = true
```

Or Do Both

```scala
object PeopleAgain {

  case class Person(
      firstName: String,
      lastName: String,
      age: Int) {

    val fullName = s"${firstName} ${lastName}"
  }

  def celebrateBirthdayOf(p: Person): Person = p.copy(age = p.age + 1)
  def fromSameFamily(p1: Person, p2: Person) = p1.lastName == p2.lastName

  implicit class PersonOps(p: Person) {
    def celebrateBirthday() = celebrateBirthdayOf(p)
    def sameFamilyAs(other: Person) = fromSameFamily(p, other)
  }

}
```

Usage:

```scala
scala> import PeopleAgain._
import PeopleAgain._

scala> val p1 = Person("John", "Doe", 43)
scala> val p2 = Person("Jane", "Doe", 43)

scala> fromSameFamily(p1, p2)
res0: Boolean = true

scala> p1.sameFamilyAs(p2)
res1: Boolean = true
```

Built-in Collection Types

- `LinearSeq[+T]`
  - `List[+T]`, `Stream[+T]`, `Queue[+T]`
- `IndexedSeq[+T]`
  - `Vector[+T]`, `Range`
- `Set[T]`
  - `HashSet[T]`, `SortedSet[T]`, `BitSet`, `ListSet[T]`
- `Map[K,+V]`
  - `HashMap[K,+V]`, `SortedMap[K,+V]`, `ListMap[K,+V]`

Characteristics:

- immutable
- covariant where appropriate
- unified operations
  - `map`, `flatMap`, `filter`, `find`, `drop`, `fold` ...
- uniform return type

Built-in Collection Types

Immutable

```
scala> val l1 = List(1,2,3,4,5,6)
l1: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> val l2 = 0 :: l1
l2: List[Int] = List(0, 1, 2, 3, 4, 5, 6)

scala> l1
res14: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> l2
res15: List[Int] = List(0, 1, 2, 3, 4, 5, 6)
```

Built-in Collection Types

Co-variance

```
scala> trait Animal
defined trait Animal

scala> trait Dog extends Animal
defined trait Dog

scala> val dog = new Dog {}
dog: Dog = $anon$1@3cc0d1ea

scala> val animal = new Animal {}
animal: Animal = $anon$1@76825483

scala> val dogs = List(dog, dog)
dogs: List[Dog] = List($anon$1@3cc0d1ea, $anon$1@3cc0d1ea)

scala> val animals = animal :: dogs
animals: List[Animal] = List($anon$1@76825483, $anon$1@3cc0d1ea, $anon$1@3cc0d1ea)
```

Built-in Collection Types

Co-variance

```scala
sealed trait MyList[+T] {
  def head: Option[T]
  def cons[TT >: T](t: TT): MyList[TT]

  // synonym for cons
  def ::[TT >: T](t: TT) = cons(t)
}

case object MyNil extends MyList[Nothing] {
  def head = None
  def cons[T](t: T) = Cons[T](t, MyNil)
}

case class Cons[T](t: T, tail: MyList[T]) extends MyList[T] {
  def head = Some(t)
  def cons[TT >: T](t: TT) = Cons[TT](t, this)
}
```

Usage:

```
scala> val l1 = dog :: dog :: MyNil
l1: MyList[Dog] = Cons($anon$1@32535909,Cons($anon$1@32535909,MyNil))

scala> val l2 = animal :: l1
l2: MyList[Animal] = Cons($anon$1@66441e91,Cons($anon$1@32535909,Cons($anon$1@32535909,MyNil)))
```

Built-in Collection Types

Unified Operations

```
def bigChain(t: Traversable[Int]) = t.filter(_ % 2 == 0).
                                      map(_ * 3).
                                      map(_.toString).
                                      drop(2).
                                      reduce(_ + _)
bigChain: (t: Traversable[Int])String

scala> bigChain(List(1,2,3,4,5,6,7))
res7: String = 18

scala> bigChain(Vector(1,2,3,4,5,6,7))
res8: String = 18

scala> bigChain(Set(1,2,3,4,5,6,7))
res9: String = 18
```

Built-in Collection Types

Uniform Return Type

```
scala> List(1, 2, 3) map (_ + 1)
res0: List[Int] = List(2, 3, 4)

scala> Set(1, 2, 3) map (_ * 2)
res0: Set[Int] = Set(2, 4, 6)
```

Algebraic Data Types

```scala
sealed trait Expr
case class Constant(v: Int) extends Expr
case class Add(left: Expr, right: Expr) extends Expr

object Expr {

  def eval(e: Expr): Int = e match {
    case Constant(value) => value
    case Add(l, r)       => eval(l) + eval(r)
  }

  def simplify(e: Expr): Expr = e match {
    case Add(Constant(0), r) => simplify(r)
    case Add(l, Constant(0)) => simplify(l)
    case Add(l, r)           => Add(simplify(l), simplify(r))
    case otherwise           => otherwise
  }

}
```

Data Structures

Round-up:

- different options available for data-representation
  - immutability encouraged but not necessary
- different options available for where to put data manipulation
  - largely a *style* decision
- sophisticated collections library to help along the way

Control Structures

Expressions

Everything is an expression

```
// if statements
scala> if (true) "foo" else "bar"
res3: String = foo

// pattern matching
scala> :paste
// Entering paste mode (ctrl-D to finish)

"foo" match {
  case "foo" => true
  case _     => false
}
res4: Boolean = true

// things that appear to be statements have a
// return type:
scala> val meaningless = {var s = ""}
meaningless: Unit = ()
```

Loops

```
// Imperative style looping
// Entering paste mode (ctrl-D to finish)

var i = 0
while(i < 10) { i += 1 }

// Exiting paste mode, now interpreting.

i: Int = 10
```

Loops

```
// Side-effectful iteration
scala> List(1,2,3).foreach(println)
1
2
3
```

Folding

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

scala> l.foldLeft(0) { _ - _ }
res13: Int = -6

scala> l.foldRight(0) { _ - _ }
res14: Int = 2
```

Recursion

```scala
object Fact {
  def fact(n: Int) = {

    @annotation.tailrec
    def factAcc(n: Int, acc: Int): Int = n match {
      case 0 => acc
      case 1 => acc
      case n => factAcc(n-1, acc*n)
    }

    factAcc(n, 1)
  }
}
```

For-comprehensions

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

val result = for {
  list <- lists
} yield list.length

// List(3,2,1,4)
```

For-comprehensions

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

val result = for {
  list <- lists
  if list.length > 1
  x <- list
} yield x

// List(1,2,3,4,5,   7,8,9,10)
```

For-comprehensions

Syntatic sugar for composing together `map()`, `flatMap()` and `filter()`

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

////////////////////////////////////////////////////////

for { list <- lists } yield list.length

// equivelant to
lists.map { _.length }

////////////////////////////////////////////////////////

for {
  list <- lists
  if list.length > 1
  x <- list
} yield (x*2)

//equivelant to
lists.filter(_.length > 1)
     .flatMap { list => list.map { _ * 2 } }
```

Control Structures

Round-up

- encourages *expression oriented* programming
- recursive functions may blow the stack if the recursive call is not in tail-position.
- can still write imperative code where needed
- for-comprehensions are just sugar

Functions

Function Types

```
String => Int === Function1[String,Int]

def applyToTest(f: String => Int) = f("test")
def applyToTestAlternative(f: Function1[String,Int]) = f("test")
```

Function Values

```scala
def strLength = (s: String) => s.length
def strSum    = (s: String) => s.map(_.toInt).sum

applyToTest(strLength)  // 4
applyToTest(strSum)     // 448 (obviously!)
```

Composing Functions

```scala
def double = (i: Int) => i * 2
val composed = strLength andThen double
val alt      = double compose strLength

composed("test")    // 8
alt("test")         // 8
```

Curried Form

```
def inCurriedForm(s1: String)(s2: String) = (s1 + " " + s2).length

applyToTest(inCurriedForm("one"))    // 8 == ("one" + " " + "test").length
```

Partially Applied

```scala
def takeLengthOf3Args(s1: String, s2: String, s3: String) = (s1+s2+s3).length
def stringToInt = takeLengthOf3Args("one", _: String, "three")

applyToTest(stringToInt) // 12
```

Partial Functions

Functions for which not all inputs are valid.

```
def recip: PartialFunction[Double, Double] = {
  case x if (x != 0.0) 1.0/x
}

recip.isDefinedAt(2.0)  // true
recip(2.0)              // 0.5

recip.isDefinedAt(0.0)  // false

recip.lift(0.0)         // None
recip.lift(2.0)         // Some(0.5)
```

Functions

Round-up

- functions are first-class objects, with use of higher-order functions the norm
- type inference doesn't always do the right thing, or needs some help
- generally, functions are written in their tupled form, and partially applied as necessary.
  - one advantage of the curried form is in type inference
- partially applied functions useful if the function is not in a curried form

Handling Failure

Exceptions

```scala
object ExceptionExample {

  def trySomething(): Int = {
    try {
      throw new java.io.IOException("Uh Oh")
    } catch {
      case (e: java.io.IOException) => 0
      case (e: ArithmeticException) => 1
    }
  }

  def trySomethingElse(): Int = {
    try {
      throw new ArithmeticException("Uh Oh")
    } catch handleIOException orElse handleMathException
  }

  private def handleIOException: PartialFunction[Throwable, Int] = {
    case (e: java.io.IOException) => 0
  }

  private def handleMathException: PartialFunction[Throwable, Int] = {
    case (e: ArithmeticException) => 1
  }

}
```

Option[T]

```scala
object OptionExample {

  private val people = Map(
    0 -> Person("John", "Doe", 43),
    1 -> Person("Jane", "Doe", 44),
    2 -> Person("John", "Smith", 54)
  )

  def findPerson(id: Int): Option[Person] = people.get(id)
  def findRelated(person: Person): Option[Person] = {
    people.values.filter(_.lastName == person.lastName)
                 .filter(_ != person)
                 .headOption
  }
}


findPerson(0).map(_.age)    // Some(43)
findPerson(3).map(_.age)    // None

/////////////////////////////////////////

for {
  p1 <- findPerson(0)
  related <- findRelated(p1)
} yield (p1, related)       // Some((Person(John,Doe,43),Person(Jane,Doe,44)))

/////////////////////////////////////////

for {
  p1 <- findPerson(2)
  related <- findRelated(p1)
} yield (p1, related)       // None
```

Try[T]

Like Option[T], but carries a *reason* for the failure

```
import scala.util.{Try, Success, Failure}

object TryExample {

  val service = OptionExample

  def findPerson(id: Int): Try[Person] = {
    Try {
      service.findPerson(id).getOrElse {
        throw new RuntimeException(s"Couldn't find: ${id}")
      }
    }
  }

  def findRelated(person: Person): Try[Person] = {
    Try {
      service.findRelated(person).getOrElse {
        throw new RuntimeException(s"Couldn't find Person related to ${person}")
      }
    }
  }
}
```

Try[T]

```
findPerson(0).map(_.age) // Success(43)
findPerson(3).map(_.age) // Failure(java.lang.RuntimeException: Couldn't find: 3)

/////////////////////////////////////////

for {
  p1 <- findPerson(0)
  related <- findRelated(p1)
} yield (p1, related)   // Success((Person(John,Doe,43),Person(Jane,Doe,44)))

/////////////////////////////////////////

for {
  p1 <- findPerson(2)
  related <- findRelated(p1)
} yield (p1, related)

// Failure(java.lang.RuntimeException: Couldn't find Person related to Person(John,Smith,54))
```

Note that the user-code is identical to the `Option[T]` example.

Handling Failure

Round-up

- Can fall back to throwing exceptions
- Exceptions are unchecked, unlike Java
- Support for more functional approaches works with for-comprehensions

Abstractions

Type Classes

First, a detour to implicits...

Implicits

```scala
trait Config { def port = 8080 }   // some sort of context

object ImplicitExample {
  def doSomething(i: Int)(implicit c: Config) {
    println(s"Doing something on port ${c.port}")
  }
}
```

Usage:

```
scala> ImplicitExample.doSomething(1)
<console>:8: error: could not find implicit value for parameter c: Config
              ImplicitExample.doSomething(1)

scala> implicit val config1 = new Config { }
config1: Config = $anon$1@3bc2ed06

scala> ImplicitExample.doSomething(1)
Doing something on port 8080

scala> implicit val config2 = new Config { override def port = 8181 }
config2: Config = $anon$1@1ec4330f

scala> ImplicitExample.doSomething(1)
<console>:10: error: ambiguous implicit values:
 both value config1 of type => Config
 and value config2 of type => Config
 match expected type Config
              ImplicitExample.doSomething(1)

scala> ImplicitExample.doSomething(1)(config2)
Doing something on port 8181
```

Implicits

- Simple light-weight dependency injection
- Safe (unambiguous)
- Locally scoped

Also, the *key* to type-classes in Scala

Defining a Typeclass

Using a typeclass

```
import Monoid._
def flatten[T](ts: List[T])(implicit ev: Monoid[T]): T = ts match {
  case Nil       => ev.zero
  case (x :: xs) => ev.mappend(x, flatten(xs))
}

flatten(List("123", "abc", "def"))                    // "123abcdef"
flatten(List(None, Option("some"), Option("values"))) // Some("somevalues")
```

Synactic Sugar

So popular, that syntactive sugar has been introduced:

```
def flatten[T: Monoid](ts: List[T]) = ???
```

Re-visit Try/Option

```scala
import scala.language.higherKinds

import scala.util.Try

import scalaz._
import scalaz.syntax.monad._
import scalaz.std.option._
import scalaz.contrib.std.utilTry._

abstract class PeopleService[M[+_]: Monad] {
  def findPerson(id: Int): M[Person]
  def findRelated(person: Person): M[Person]

  def findRelatedTo(id: Int): M[Person] = {
    for {
      p <- findPerson(id)
      related <- findRelated(p)
    } yield related
  }
}

object OptionPeopleService extends PeopleService[Option] {
  lazy val service = OptionExample
  def findPerson(id: Int) = service.findPerson(id)
  def findRelated(person: Person) = service.findRelated(person)
}

object TryPeopleService extends PeopleService[Try] {
  lazy val service = TryExample
  def findPerson(id: Int) = service.findPerson(id)
  def findRelated(person: Person) = service.findRelated(person)
}
```

Handling Effects and State

Handling Effects And State

Ran out of time. . . .