Comp 311
Functional Programming

Nick Vrvilo, Two Sigma Investments
Robert “Corky” Cartwright, Rice University

November 14, 2019

Announcements

- Homework 4 is due Thursday
- Final exam info is on the class calendar

Some Additional Scala Features

Scripting in Scala

- Scala is designed for building large-scale systems
- It also scales down to small scripts:
    - In a single file, we can place class definitions, function definitions, and even top-level expressions

Scripting in Scala

- In a single file `hello.scala`, write:

```
println(“Hello, scripting world!”)
```

- From the command-line (in an environment where scala has been installed):

```
scala hello.scala
```

Scripting in Scala

- Command-line arguments are available via a global array named `args`:

```
println(“Hello, ” + args(0) + “!”)
```

Scripting in Scala

- At the shell:

```
scala hello.scala Owls
```

- And the result is:

```
Hello, Owls!
```

Scripting in Scala

- On Unix, you can run a Scala script directly from the shell by putting a *shebang* at the top of your script:

```
#!/usr/bin/env scala

println("hello“)
```

- Then make the file executable (let's name the file `hello`):

```
chmod u+x hello
```

Scala Applications The "Java" Way

- To compile a stand-alone Scala application, you can put the driver into a singleton object with a `main` method

Scala Applications

- Any singleton object might contain a main method that takes an argument of type `Array[String]`:

```
package edu.rice.cs.comp311.lectures.lecture22

object ArgLengths {
  def main(args: Array[String]): Unit = {
    for (arg <- args)
      println(arg + ": " + arg.length)
  }
}
```

Scala Applications
The “Scala” Way

- To compile a stand-alone Scala application, you can put the driver into a singleton object with the `App` trait

- All code in the body of the object (i.e., the “constructor” code) is run when the app is launched

Scala Applications

- Any singleton object might contain a main method that takes an argument of type `Array[String]`:

```
package edu.rice.cs.comp311.lectures.lecture22

object ArgLengths extends App {
  for (arg <- args) {
    println(arg + ": " + arg.length)
  }
}
```

For loops (no *yeild* keyword) are only for side-effects.
Just syntactic sugar for the *foreach* method.

Scala Applications

- Any singleton object might contain a main method that takes an argument of type `Array[String]`:

```
package edu.rice.cs.comp311.lectures.lecture22

object ArgLengths extends App {
  args foreach { arg =>
    println(arg + ": " + arg.length)
  }
}
```

Scala Applications

- Compile using `scalac` or `fsc`
  - `scalac` will recompile all referenced jars, files,...
    - Therefore, it can be slow
  - `fsc` starts a process the first time it is run that memoizes compilation of referenced files

Scala Applications

- Execute a compiled classfile using the `scala` command
    - Include the full path name

```
scala edu.rice.cs.comp311.lectures.lecture22.ArgLengths
```

Fields in Non-Case Classes

- constructor of a class is a function:
  - When it is called, the enclosing environment is extended and an object is returned, as defined by the body of the class

Fields in Non-Case Classes

- A natural consequence:
  - The arguments to a constructor call are not directly accessible outside the object that is returned from the call
- To make a parameter accessible, define a field
- Case classes automatically define a field for every constructor parameter

The Follow Code Will Not Pass Type Checking

```
class Rational(numerator: Int, denominator: Int) {
  def +(that: Rational) =
    new Rational(numerator * that.denominator +
                 that.numerator * denominator,
                 denominator * that.denominator)
}
```

Declaring the Fields Explicitly Fixes The Problem

```
class Rational(n: Int, d: Int) {
  val numerator = n
  val denominator = d

  def +(that: Rational) =
    new Rational(numerator * that.denominator +
                 that.numerator * denominator,
                 denominator * that.denominator)
}
```

Auxiliary Constructors

- Scala allows for multiple constructor declarations
- Additional constructors are defined as methods with name `this`
- The first action of an auxiliary constructor must be to invoke another constructor
  - Only constructors defined earlier in the class definition are in scope

Auxiliary Constructors

```
class Rational(n: Int, d: Int) {
  val numerator = n
  val denominator = d

  def this(n: Int) = this(n, 1)

  def +(that: Rational) =
    new Rational(numerator * that.denominator +
                 that.numerator * denominator,
                 denominator * that.denominator)
}
```

Auxiliary Constructors

```
class Rational(
      val numerator: Int,
      val denominator: Int) {

  def this(n: Int) = this(n, 1)

  def +(that: Rational) =
    new Rational(numerator * that.denominator +
                 that.numerator * denominator,
                 denominator * that.denominator)
}
```

Companion Objects

- A class can be given a *companion object*:
  - A singleton object definition with the same name
  - Must be defined in the same file as the class
  - The object and class share private members

Companion Objects and Factory Methods

- Companion objects are well-suited for defining factory methods:

```
object Rational {
  def apply(n: Int, d: Int) =
    if (d != 0) new Rational(n, d)
    else throw new Error("Given a zero denominator")
}
```

Private Primary Constructors

- Primary constructors can be hidden by prefixing them with the keyword private:

```
class Rational private(n: Int, d: Int) {
  val numerator = n
  val denominator = d

  def this(n: Int) = this(n, 1)

  def +(that: Rational) =
    new Rational(numerator * that.denominator +
                 that.numerator * denominator,
                 denominator * that.denominator)
}
```

Private Constructors and Companion Objects

```
> Rational(1,1)          // ok
> Rational(1,0)          // error
> new Rational(1,2)      // error
> new Rational(2)        // ok
```

Extractors

Extractors

- It is possible to control how an object will interact with pattern matching through the use of *extractors*

- Extractors are objects that define an unapply method, which takes an object and returns an option of one or more elements

Extractors

```
object Rational {
  def apply(n: Int, d: Int) = {
    if (d != 0) new Rational(n, d)
    else throw new Error("Given a zero denominator")
  }

  def unapply(q: Rational): Option[(Int, Int)] = {
    Some((q.numerator, q.denominator))
  }
}
```

Extractors

- An unapply method is called in a pattern by prefixing the name of the extractor object followed by a tuple of expected elements

- If the unapply method returns Some((x1,...xN)) and the arity of the tuple (x1,...xN) matches the number of bound variables in the pattern, we have a match

Extractors

```
class Rational private(n: Int, d: Int) {
  val numerator = n
  val denominator = d

  def +(that: Rational) = {
    that match {
      case Rational(n2,d2) =>
        Rational(n * d2 + n2 * d,
                 d * d2)
    }
  }
}
```

Case Classes Revisited

- We are now in a position to better explain what a case class definition is given implicitly:
  - Immutable fields for every parameter
  - Structural `equals` and `hashCode` methods
  - A structural `toString` method
  - A companion object with `apply` and `unapply` methods
  - A `copy` method with parameters for each constructor parameter, defaulted to the field values of the receiver

Extractors vs Case Classes

- Explicit extractors are more verbose than using case classes
- However, they have advantages of their own:
  - separates implementation from pattern matching
  - can deconstruct objects outside of their class definitions
  - can perform more sophisticated deconstruction
    - e.g. regular expression matching on strings

Extractors vs Case Classes

- Case classes also have many advantages:
  - Conciseness
  - Performance: Scala compiler optimizes patterns with case classes aggressively

Implicit Conversions

Implicit Defs

```
case class Coordinate(x: Int, y: Int)

implicit def pair2coord(pair: (Int, Int)) = {
  Coordinate(pair._1, pair._2)
}

def action(coord: Coordinate) = "OK"

action(1->2) // ↦ "OK"
```

Example: RegexParsers

```
object MyParser extends RegexParsers {
  // RegexParsers.literal converts string to Parser
  def ok: Parser[String] = "OK"

  // RegexParsers.regex converts Regex to Parser
  def yesNo: Parser[String] = "[Yy]es|[Nn]o".r
}
```

Implicit Classes

```
implicit class TitleStr(str: String) {
  def titleCase: String = {
    str.split(' ').map(_.capitalize).mkString(" ")
  }
}
```

```
"hello world".titleCase
↦ "Hello World"
```

Extension Methods *a la* Implicit Value Types

```
implicit class TitleStr(val str: String) extends AnyVal {
  def titleCase: String = {
    str.split(' ').map(_.capitalize).mkString(" ")
  }
}



"hello world".titleCase
↦ "Hello World"
```

Implicit Conversions

https://docs.scala-lang.org/tour/implicit-conversions.html

Value Classes

https://docs.scala-lang.org/overviews/core/value-classes.html

Lazy Vals

- Scala supports three types of bindings:
  (this is a review or a preview of Comp 411)

```
val i = 1          // by-value binding

def j = 2          // by-name binding

lazy val j = 2     // by-need binding
```

- Lazy vals are useful for values that you want to compute once, but are either expensive (so you don't want to make it a val), or have complex mutual dependencies (like parser combinators).

Type Aliases

- Scala supports declaring type aliases

```
type T = Map[Int, String]
type A[B] = List[B]
```

- Type aliases are members of classes, which can be declared abstract and overridden in subclasses. (Used for things like dependency injection.)

- Different from a type parameter because it can be declared private inside the class.