

Comp 311

Functional Programming

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

September 17, 2019

Homework 1

- Please submit your homework via the *SVN* / `turnin` system, in a folder named `hw_1`
- The specific files to submit are defined in the description for each assignments
- For each section, please turn in only your final program resulting from completion of the section

Please Restrict Your Homework Submission
to Features Covered in Class

Current Core Scala Features

- (case) object
- (case) class
- val
- if / else
- match / case
- require, ensuring
- Int, Double, String
- Array[T], Tuples
- Arithmetic operators
- (In)equality operators
- Logical and / or
- assert
- λ -expressions (ensuring)
- Plus the stuff from today!

Please Restrict Your Homework Submission to Features Covered in Class

This should be the only import statements you need:

```
import org.scalatest._
```

(or equivalent imports auto-generated by your IDE for
your ScalaTest test class)

Methods and Operators

Syntactic Sugar For Binary Methods

- We refer to methods that take one parameter (in addition to the receiver) as *binary methods*

```
case class Coordinate(x: Int, y: Int) {  
  def magnitude() = x*x + y*y  
  
  def add(that: Coordinate) =  
    Coordinate(x + that.x, y + that.y)  
}
```

Syntactic Sugar For Binary Methods

```
Coordinate(1,2).add(Coordinate(3,4))
```

↳

```
Coordinate(4,6)
```


Syntactic Sugar For Binary Methods

- We can elide the dot in method calls on binary methods
- We can also elide the enclosing parentheses around the sole argument

Syntactic Sugar For Binary Methods

`Coordinate(1,2) add Coordinate(3,4)`

\mapsto

`Coordinate(4,6)`

Operator Symbols

- Scala allows the use of operator symbols in method names
- In fact, operators are simply methods in Scala

`1 + 2` \mapsto `3`

`1.+ (2)` \mapsto `3`

Coordinate Custom +

```
case class Coordinate(x: Int, y: Int) {  
  def magnitude() = x*x + y*y  
  
  def +(that: Coordinate) =  
    Coordinate(x + that.x, y + that.y)  
}
```

Coordinate Custom +

`Coordinate(1,2) + Coordinate(3,4)`

`↳`

`Coordinate(4,6)`

Requires Clauses on Class Constructors

```
case class Name(field1: Type1, ..., fieldN: TypeN) {  
  require (boolean-expression)  
  ...  
}
```

- Checked on every constructor call
- Because case class instances are immutable, this ensures the property holds for the lifetime of an instance

Equals on Case Classes

- The equals method on a case class instance checks for structural equality with its argument:

```
Rational(4,6).equals(Rational(4,6)) ↪
```

```
true
```

Equals on Case Classes

- Note that equals is a binary method, and so we can also write this expression as:

`Rational(4,6) equals Rational(4,6) ⇨`

`true`

Equals on Case Classes

- The `==` operator in Scala, unlike Java, delegates to the `equals` method:

```
Rational(4,6) == Rational(4,6) ⇨  
true
```

Equals on Case Classes

- Of course, the built in equals method does not check for mathematical equality:

```
Rational(4,6) == Rational(2,3) ↪
```

```
false
```

Equals on Case Classes

- Why is this definition of equality acceptable on case classes?
- What other definition is available to us?

```
Rational(4,6) == Rational(2,3) ↪
```

```
false
```

Calling and Defining Parameterless Methods Without Parentheses

```
def toString() = { ... }
```

vs.

```
def toString = { ... }
```

Calling and Defining Parameterless Methods Without Parentheses

```
Rational(4,6).toString()
```

vs.

```
Rational(4,6).toString
```

The Uniform Access Principle

- Client code should not be affected by whether an attribute is defined as a field or a method
 - Only applies to *pure* (side-effect free) methods
 - Can be strange even for some pure methods (what are some examples?)

Abstract Datatypes

Abstract Datatypes

- Often, we wish to abstract over a collection of compound datatypes that share common properties
- For example, we might wish to define an abstract datatype for shapes, with separate case classes for each of several shapes
- For this purpose, we define an *abstract class* and use *subclassing*

Abstract Datatypes

```
abstract class Shape
case class Circle(radius: Double) extends Shape
case class Square(side: Double) extends Shape
case class Rectangle(height: Double, width: Double) extends Shape
```

Abstract Methods

```
abstract class Shape {  
  def area: Double  
}
```

```
case class Circle(radius: Double) extends Shape {  
  val pi = 3.14  
  def area = pi * radius * radius  
}
```

```
case class Square(side: Double) extends Shape {  
  def area = side * side  
}
```

```
case class Rectangle(length: Double, width: Double)  
extends Shape {  
  def area = length * width  
}
```

One Method to Rule Them All

```
abstract class Shape {  
  val pi = 3.14  
  def area: Double = this match {  
    case Circle(radius) => pi * radius * radius  
    case Square(side) => side * side  
    case Rectangle(width, height) => width * height  
  }  
}
```

Applying a Class Method Revisited

- To reduce the application of a method:

$C(v_1, \dots, v_k).m(\text{arg}_1, \dots, \text{arg}_N)$

- Reduce the receiver and arguments, left to right
- ~~Reduce the body of m~~ , replacing constructor parameters with constructor arguments and method parameters with method arguments

Applying a Class Method Revisited

- To reduce the application of a method:

$C(v_1, \dots, v_k).m(\text{arg1}, \dots, \text{argN})$

- Reduce the receiver and arguments, left to right
- **Find the body of m in C and reduce to that,** replacing constructor parameters with constructor arguments and method parameters with method arguments

The Body of m

- To find the body of method m in type C :
 - Find the definition of m in the body of C , if it exists
 - Otherwise, find the body of m in the immediate superclass of C

Abstract Datatype

Example: Option

The Option Class

- The `Option` class is a collection of zero or one items.
- The parameterized type `Option[T]` denotes a collection of at most one object with type `T`.
- The `Some[T]` subclass represents the non-empty case.
- The `None` object represents the empty case.

Option Implementation

```
abstract class Option[T] {  
  def get: T  
  def isEmpty: Boolean  
  def nonEmpty: Boolean  
}
```

```
case class Some[T](x: T) extends Option[T] {  
  def get = x  
  def isEmpty = false  
  def nonEmpty = true  
}
```

```
case object None extends Option[Nothing] {  
  def get: T =  
    throw new java.util.NoSuchElementException()  
  def isEmpty = true  
  def nonEmpty = false  
}
```

Design Templates for Abstract Datatypes

Case 1

**We Expect Few New Functions
But Many New Variants**

Abstract Methods

```
abstract class Shape {  
  def area: Double  
}
```

```
case class Circle(radius: Double) extends Shape {  
  val pi = 3.14  
  def area = pi * radius * radius  
}
```

```
case class Square(side: Double) extends Shape {  
  def area = side * side  
}
```

```
case class Rectangle(length: Double, width: Double)  
extends Shape {  
  def area = length * width  
}
```

Case Two

**We Expect Many New Functions
But Few New Variants**

One (Pattern Matching) Method to Rule Them All

```
abstract class Shape {  
  val pi = 3.14  
  def area: Double = this match {  
    case Circle(radius) => pi * radius * radius  
    case Square(side) => side * side  
    case Rectangle(width, height) => width * height  
  }  
}
```

Case 2: We Expect Many New Functions But Few New Variants

- This is a case that traditional functional programming handles well
- Classic example domains: Compilers, theorem provers, numeric algorithms, machine learning
- Declare a top-level function with cases for each data variant

a.k.a., The Visitor Pattern

We Can Define Arbitrary Functions Without Modifying Data Definitions

```
def makeLikeFirst(shape0: Shape, shape1: Shape) = {  
  (shape0, shape1) match {  
    case (Circle(r), Square(s)) => Circle(s)  
    case (Circle(r), Rectangle(l,w)) => Circle((l+w)/2)  
  
    case (Square(s), Circle(r)) => Square(r)  
    case (Square(s), Rectangle(l,w)) => Square((l+w)/2)  
  
    case (Rectangle(l,w), Circle(r)) => Rectangle(r,r)  
    case (Rectangle(l,w), Square(s)) => Rectangle(s,s)  
  
    case _ => shape1  
  }  
}
```


But A New Data Variant Requires Us To Modify All Functions Over the Datatype

```
val pi = 3.14

def area(shape: Shape) = {
  shape match {
    case Circle(r) => pi * r * r
    case Square(x) => x * x
    case Rectangle(x,y) => x * y
    case Triangle(b,h) => b*h/2
  }
}
```

But A New Data Variant Requires Us To Modify All Functions Over the Datatype

```
def makeLikeFirst(shape0: Shape, shape1: Shape) = {  
  (shape0, shape1) match {  
    case (Circle(r), Square(s)) => Circle(s)  
    case (Circle(r), Rectangle(l,w)) => Circle((l+w)/2)  
    case (Circle(r), Triangle(b,h)) => Circle(b)  
  
    case (Square(s), Circle(r)) => Square(r)  
    case (Square(s), Rectangle(l,w)) => Square((l+w)/2)  
    case (Square(s), Triangle(b,h)) => Square(b+h/2)  
  
    case (Rectangle(l,w), Circle(r)) => Rectangle(r,r)  
    case (Rectangle(l,w), Square(s)) => Rectangle(s,s)  
    case (Rectangle(l,w), Triangle(b,h)) => Rectangle(b,h)  
  
    // plus all the cases for Triangle on the left (omitted)  
    case _ => shape1  
  }  
}
```

Sealed Data Types

- Adding the **sealed** keyword to an abstract type indicates that all subclasses of that type are declared in the current compilation unit.
- Provides extra information to the compiler for optimizations and diagnostics

```
sealed abstract class Shape  
case class Square(length: Double) extends Shape  
case class Circle(radius: Double) extends Shape  
case class Triangle(base: Double, height: Double)  
    extends Shape
```

Sealed Data Types

```
object Math {  
  val pi = 3.141592653589793  
}
```

```
sealed abstract class Shape {  
  def area: Double = this match {  
    // case Square(x) => x * x  
    case Circle(r) => Math.pi * r * r  
    case Triangle(b, h) => 0.5 * b * h  
  }  
}
```

warning: match may not be exhaustive.
It would fail on the following input: Square(_)
def area: Double = this match {