

Unifying Object-Oriented and Functional Programming

Jorge Ortiz
CS94SI

“Scala goes further than all other well-known languages in fusing object-oriented and functional programming.”

- Martin Odersky, creator of Scala

What is OOP?

- Main goals:
 - Abstraction
 - Code reuse
 - Unifying data and behavior
 - Controlling complexity

What is OOP?

- It attempts to achieve these goals with:
 - Modularity
 - Inheritance
 - Encapsulation
 - Polymorphism

What is FP?

- Most restrictive definition: *Pure* functional languages = no side effects
- More general definition: “Impure” languages have some functional aspects in common

What is FP?

- FP can include:
 - garbage collection
 - parametrized types
 - pattern matching
 - functions as first-class values
 - lazy evaluation

"We were after the C++ programmers.
We managed to drag a lot of them about
halfway to Lisp."

- Guy Steele, co-author of the Java spec

FP + OOP

- Some FP concepts have found good homes in OO languages
- Garbage collection (Smalltalk, Java, C#)
- Parametrized types (C++, Java, C#)
- Functions as first-class values (Smalltalk, C#, Java?)

Pattern Matching

Pattern Matching?

- Few languages support both Object-Oriented Programming and Pattern Matching
- Why?
- OOP and PM deal with very different problem domains

PM + OOP?

- OOP is good for problems where:
 - Fixed set of methods to be supported
 - Extensible set of objects supporting them
 - Good example: GUI library

PM + OOP?

- Pattern matching is good for problems:
 - Fixed set of data types to be supported
 - Extensible set of methods on those types
 - Good example: Compiler

PM + OOP?

- Why not include PM in OO languages?
- Three OO criticism of pattern matching:
 - Unnecessary
 - Not extensible
 - Breaks encapsulation

PM + OOP?

- **Unnecessary** – use *visitor design pattern*

```
interface Visitor {  
    void visit(Sum sum);  
    void visit(Product product);  
    void visit(Constant constant);  
    ...  
}
```

```
class Sum {  
    public void accept(Visitor visitor) {  
        visitor.visit(this);  
    }  
    ...  
}
```

- Challenge: Do HW#3 with visitors
 - Verbose
 - Doesn't allow nested patterns

PM + OOP?

- **Not extensible**, algebraic data types have a fixed number of cases
- This is true in some functional languages
- Not true in Scala
 - case classes can both inherit and be inherited from

PM + OOP?

- **Breaks encapsulation**
 - In most functional languages, cases are a thin wrapper around data
 - OOP is about hiding data, pattern matching is about exposing it
 - In Scala, case classes can have private members too
 - Consider constructor parameters part of the contract for the class

PM + OOP in Scala

- Case classes
 - Scala's version of algebraic data types
- Extractors
 - A way for pattern matching to cohabit with OO data abstraction

Extractors

- Extractors are used for pattern matching on non-case classes
- Useful to think of them as user-defined *views* on or into objects

Extractors

```
val s: String = "jorge@gmail.com"
s match {
  case EMail(name, domain) =>
    println(name + " AT " + domain)
  case _ =>
    println("Not an e-mail address")
}

// prints "jorge AT gmail.com"
```

Extractors

```
val s: String = "Some Junk String"
s match {
  case EMail(name, domain) =>
    println(name + " AT " + domain)
  case _ =>
    println("Not an e-mail address")
}

// prints "Not an e-mail address"
```

Extractors: How?

```
object EMail {  
  def unapply(email: String):  
Option[(String, String)] = {  
  val parts = email split "@"  
  if (parts.length == 2)  
    Some(parts(0), parts(1))  
  else  
    None  
  }  
}
```

Extractors: How?

```
case class Cartesian(x: Double, y: Double)
```

```
object Polar {  
  def unapply(z: Cart):  
    Option[(Double, Double)] =  
    ...  
}
```

```
Cart(1, 1) match {  
  case Polar(r, th) => ...  
    // r == sqrt(2), th == Pi / 4  
}
```

Extractors

- Provide user-defined *views* on objects
- Defined with an `unapply` method in a singleton object (usually the companion object for a class, but not necessarily so)
- Decouples types from pattern matching
- Allows pattern matching to cohabit with data abstraction – extractor can provide same view despite implementation changes

Functions as Objects, Objects as Functions

Functions

- Functions in Scala:
 - Inherit from the `FunctionN` trait (where `N` is number of arguments)
 - Define an `apply` method
 - Can be subclassed in powerful ways

Functions

```
trait Function0[+R] {  
  def apply(): R  
}
```

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

```
trait Function2[-T1, -T2, +R] {  
  def apply(v1: T1, v2: T2): R  
}
```

Partial Function

```
trait PartialFunction[-A, +B]  
  extends (A => B) {  
  // Inherits apply from Function1  
  def isDefinedAt(x: A): Boolean  
}
```

Set

```
trait Set[A]  
  extends (A => Boolean) ...
```

Array

```
class Array[A]  
  extends PartialFunction[Int, A] ...
```

Syntactic Sugar

- Any use of `obj(...)` will invoke `apply(...)` method on `obj` (even if `obj` does not inherit from `FunctionN`)
- Useful for (among other things) factory methods in companion objects

Apply

```
object EMail {  
  def unapply...  
  def apply(name: String, domain: String):  
String =  
  name + "@" + domain  
}
```

```
EMail("jorge", "gmail.com")  
  // == "jorge@gmail.com"
```

Apply

```
case class Cartesian(x: Double, y: Double)
```

```
object Polar {  
  def unapply...  
  def apply(r: Double, th: Double):  
    Cartesian =  
      Cartesian(r * cos(th), r * sin(th))  
}
```

```
Polar(sqrt(2), Pi/4)  
  // == Cartesian(1.0, 1.0)
```

Functions + Scala Compiler

```
val square = (x: Int) => x*x
```

```
square(2)
```

Functions + Scala Compiler

```
val square = (x: Int) => x*x
```

```
square(2)
```

Functions + Scala Compiler

```
val square = new Function1[Int, Int] {  
  def apply(x: Int): Int = x*x  
}
```

```
square(2)
```

Functions + Scala Compiler

```
val square = new Function1[Int, Int] {  
  def apply(x: Int): Int = x*x  
}
```

square(2)

Functions + Scala Compiler

```
val square = new Function1[Int, Int] {  
  def apply(x: Int): Int = x*x  
}
```

```
square.apply(2)
```

Functions + Scala Compiler

```
val square = new Function1[Int, Int] {  
  def apply(x: Int): Int = x*x  
}
```

```
square.apply(2)
```

Functions + Scala Compiler

```
Function1<Integer, Integer> square =  
  new Function1<Integer, Integer> {  
    public Integer apply(Integer x) {  
      return x*x;  
    }  
  }
```

```
square.apply(2)
```

Functions + Scala Compiler

```
val square = (x: Int) => x*x
```

```
square(2)
```

Functions + Scala Compiler

```
val filter =  
  (c: List[A], p: A => Boolean) =>  
    ...
```

Functions + Scala Compiler

```
Function2<List<A>, Function1<A, Boolean>, List<A> > filter =  
  new Function2<List<A>, Function1<A, Boolean>, List<A> > {  
    public List<A> apply(List<A> c, Function1<A, Boolean> p) {  
      ...  
    }  
  }
```

Lazy Programming

Evaluation Strategies

- Two evaluation strategies:
 - *Eager* – evaluate expressions as soon as possible
 - *Lazy* – evaluate expressions only when they are needed
- Most languages use eager evaluation

Eager Evaluation

```
val pi = {  
    println("Hello, world!")  
    3.14159  
}
```

```
println("Goodbye, world!")  
println(pi)
```

```
// Prints: Hello, world!  
//           Goodbye, world!  
//           3.14159
```

Lazy Evaluation?

```
val pi = {  
    println("Hello, world!")  
    3.14159  
}
```

```
println("Goodbye, world!")  
println(pi)
```

```
// Prints: ???  
//      ???  
//      ???
```

Lazy Evaluation

```
lazy val pi = {  
    println("Hello, world!")  
    3.14159  
}
```

```
println("Goodbye, world!")  
println(pi)
```

```
// Prints: Goodbye, world!  
//           Hello, world!  
//           3.14159
```

Lazy Evaluation

```
lazy val pi = {  
    println("Hello, world!")  
    3.14159  
}
```

```
println(pi)  
println(pi)
```

```
// Prints: ???  
//        ???  
//        ???
```

Lazy Evaluation

```
lazy val pi = {  
    println("Hello, world!")  
    3.14159  
}
```

```
println(pi)  
println(pi) // pi got cached, not  
            // evaluated again  
// Prints: Hello, world!  
//         3.14159  
//         3.14159
```

Lazy C/C++/Java?

- C, C++, and Java have short-circuit evaluation
 - `a && b`
 - `b` only gets evaluated if `a` is true
 - `a || b`
 - `b` only gets evaluated if `a` is false

Lazy C/C++/Java?

- C, C++, and Java also have if/then, while, for

```
if (condition) {  
    block  
}
```

- Block only gets evaluated if condition is true

Lazy C/C++/Java?

- In C, C++, Java, control structures (which use a form of lazy evaluation) are defined by the language, aren't extensible

Lazy Scala

- Scala has two language constructs for lazy evaluation:
 - lazy vals
 - thunk parameters
- These features let us define our own control structures

while

```
def myWhile(cond: => Boolean)(block: => Unit): Unit = {  
  if (cond) {  
    block  
    myWhile(cond)(block)  
  }  
}
```

```
var i = 0 // print: 0  
myWhile(i < 10) { // 1  
  println(i) // 2  
  i = i + 1 // 3  
} // ...
```

while

```
def myWhile(cond: => Boolean)(block: => Unit): Unit = {  
  if (cond) {  
    block  
    myWhile(cond)(block)  
  }  
}
```

```
var i = 11  
myWhile(i < 10) {  
  println(i) // prints nothing  
  i = i + 1  
}
```