

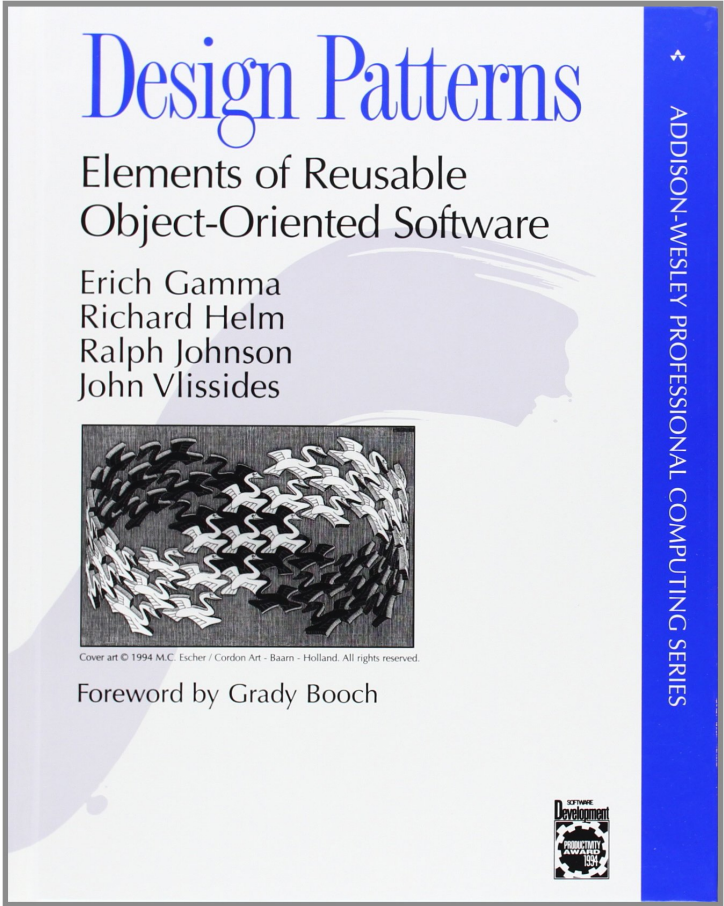
# Functional Design Patterns in Scala

@rich\_ashworth

# What we'll cover

- What is a functional design pattern?
- Examples
  - ADTs
  - Optics
  - Type Classes
  - Monoids

# Design Patterns



## THE 23 GANG OF FOUR DESIGN PATTERNS

C	Abstract Factory	S	Facade	S	Proxy
S	Adapter	C	Factory Method	B	Observer
S	Bridge	S	Flyweight	C	Singleton
C	Builder	B	Interpreter	B	State
B	Chain of Responsibility	B	Iterator	B	Strategy
B	Command	B	Mediator	B	Template Method
S	Composite	B	Memento	B	Visitor
S	Decorator	C	Prototype		

# Algebraic Data Types

# Algebraic Data Types

Sum for is-a:

```
sealed trait Language
final case object Java extends Language
final case object Python extends Language
final case object Scala extends Language
```

Product for has-a:

```
case class Identity(name: String, address: Address)
case class Address(line1: String, line2: String, postcode: String)
```

# Algebraic Data Types

Typically we combine sum and product types in an ADT:

```
sealed trait Employee
final case class Manager(who: Identity, reports: Set[Employee]) extends Employee
final case class Developer(who: Identity, primarySkill: Language) extends Employee
final case class Tester(who: Identity) extends Employee
```

# Lenses

Problem: *Manipulate state, while preserving immutability*

```
val addr = Address("10", "Main St", "NW1 6XE")  
val rich0 = Developer(Identity("Rich", addr), Java)
```

Case classes provide the copy method. Easy enough to use for changing shallow values:

```
val rich1 = rich0.copy(primarySkill = Scala)
```

# Lenses

Less readable for values that are deeply nested:

```
val rich2 = rich1.copy(  
    who = rich1.who.copy(  
        address = rich1.who.address.copy(  
            line1 = "11"))))
```



# Lenses

We can solve this a Lens!

```
case class Lens[O, V](  
  get: O => V,  
  set: (O, V) => O  
)
```

O = top-level object

V = nested value being accessed or mutated. i.e. O has-a V

# Lenses

The most important concept here is that lenses *compose*:

```
case class Lens[I, V](  
  get: I => V,  
  set: (I, V) => I  
)  
  
def compose[I, J, V](firstLens: Lens[I, J], secondLens: Lens[J, V]) =  
  Lens[I, V](  
    get = firstLens.get andThen secondLens.get,  
    set = (obj, value) =>  
      firstLens.set(obj, secondLens.set(firstLens.get(obj), value))  
  )
```

# Lenses

```
val developerIdentityLens= Lens[Developer, Identity](  
  get = _.who,  
  set = (o, v) => o.copy(who = v)  
)  
val identityAddressLens = Lens[Identity, Address](  
  get = _.address,  
  set = (o, v) => o.copy(address = v)  
)  
val developerAddressLens = compose(developerIdentityLens, identityAddressLens)  
  
val addressLine1Lens= Lens[Address, String](  
  get = _.line1,  
  set = (o, v) => o.copy(line1 = v)  
)  
val developerLine1Lens = compose(developerAddressLens, addressLine1Lens)
```

# Lenses

So rather than:

```
val firstLine = rich0.who.address.line1 // 10
val rich1 = rich0.copy(
    who = rich0.who.copy(
        address = rich0.who.address.copy(
            line1 = "11"))))
// Developer(Identity(Rich,Address(11,Main St,NW1 6XE)),Java)
```

... we can now write:

```
val firstLine = developerLine1Lens.get(rich0) // 10
val rich2 = developerLine1Lens.set(rich0, "11")
// Developer(Identity(Rich,Address(11,Main St,NW1 6XE)),Java)
```

# Optics

Lens is only one pattern in the family

See also: `Optional`, `Prism`, `Iso`, ...

We can reduce boilerplate even further using libraries

- Monocle
- Shapeless
- Scalaz

# Type Classes

Problem: *Add behaviours to our data types*

# Type Classes

Problem: *Add behaviours to our data types without changing the types*

With subtyping:

```
trait MakesNoise { def sounds: String }

object Dog extends MakesNoise {
  override def sounds = ("bark")
}

object Laptop extends MakesNoise {
  override def sounds = "whirr, beep"
}
```

# Type Classes

```
class Dog
class Laptop

trait MakesNoise[T] { def sounds: String }

object MakesNoiseInstances {
  implicit val makeNoiseDog: MakesNoise[Dog] = new MakesNoise[Dog] {
    def sounds = "bark"
  }
  implicit val makeNoiseLaptop: MakesNoise[Laptop] = new MakesNoise[Laptop] {
    def sounds = "whirr, beep"
  }
}
```



# Type Classes

Behaviour from the type class is provided to our types through the implicit scope:

```
import MakesNoiseInstances._

implicit val makeNoiseDog: MakesNoise[Dog] = new MakesNoise[Dog] {
  def sounds = "growl"
}

val dogBehaviour = implicitly[MakesNoise[Dog]]
val laptopBehaviour = implicitly[MakesNoise[Laptop]]

println(dogBehaviour.sounds) // growl
println(laptopBehaviour.sounds) // whirr, beep
```

# Functional Libraries

Cats provides definitions of a number of common type classes, instances for basic types, and syntax.

Example:

```
trait Show[A] {  
    def show(a: A): String  
}
```

```
import cats.Show
import cats.instances.all._
import cats.syntax.show._
```

```
implicitly[Show[Int]].show(42) // 42
42.show                       // 42
```

```
sealed abstract class Colour(val name: String)
object Colour {
  implicit val ColourShow = Show.show[Colour](_.name)
  object Red extends Colour("Red")
  object Blue extends Colour("Blue")
}
```

```
implicit class ColourExt(c: Colour){
  def show = Colour.ColourShow.show(c)
}
```

```
import Colour._
println(Red.show) // Red
```

# Monoid

```
trait Semigroup[A] {  
  def combine(x: A, y: A): A  
}
```

```
trait Monoid[A] extends Semigroup[A] {  
  def empty: A  
}
```

# An Example

Problem: *Convert a string of digits into a string representing an LCD display*

```
display("012345")
```

```
//  . _ .   . . .   . _ .   . _ .   . . .   . _ .  
//  | . |   . . |   . _ |   . _ |   | _ |   | _ .  
//  | _ |   . . |   | _ .   . _ |   . . |   . _ |
```

```
case class LCDDisplay(  
  firstRow: String, secondRow: String, thirdRow: String)
```

```
object LCDDisplay {  
  val digitMapping = Map(  
    0 -> LCDDisplay(  
      ". _ .",  
      "| . |",  
      "| _ |"  
    ),  
    1 -> LCDDisplay(" . . .", " . . |", " . . |"),  
    2 -> LCDDisplay(". _ .", " . _ |", "| _ ."),  
    3 -> LCDDisplay(". _ .", " . _ |", " . _ |"),  
    4 -> LCDDisplay(" . . .", "| _ |", " . . |"),  
    5 -> LCDDisplay(". _ .", "| _ .", " . _ |"),  
    6 -> LCDDisplay(". _ .", "| _ .", "| _ |"),  
    7 -> LCDDisplay(". _ .", " . . |", " . . |"),  
    8 -> LCDDisplay(". _ .", "| _ |", "| _ |"),  
    9 -> LCDDisplay(". _ .", "| _ |", " . . |")  
  )  
}
```

# Add instances for Show and Monoid for our type:

```
object LCDDisplay {  
  ...  
  
  implicit val ShowInstance =  
    Show.show[LCDDisplay](_.productIterator mkString "\n")  
  
  implicit val ConcatMonoid = new Monoid[LCDDisplay] {  
    override def empty = LCDDisplay("", "", "")  
    override def combine(l1: LCDDisplay, l2: LCDDisplay): LCDDisplay =  
      LCDDisplay(  
        l1.firstRow + " " + l2.firstRow,  
        l1.secondRow + " " + l2.secondRow,  
        l1.thirdRow + " " + l2.thirdRow)  
  }  
  
  def parse(s: String): Seq[LCDDisplay] =  
    s.map(i => digitMapping(i.asDigit))  
  
  def display(s: String) =  
    ShowInstance.show(ConcatMonoid.combineAll(parse(s)))  
}
```

We can check this works:

```
import LCDDisplay._
```

```
println(display("012345"))
```

```
//  ._. . . . ._. ._. . . ._.
//  |.| .|. .|. ._| ._| |_| |_|.
//  |_| .|. |_| ._. ._| .|. ._|
```



We can check this works:

```
import LCDDisplay._
```

```
println(display("012345"))
```

```
//  ._. . . ._. ._. . . ._.  
//  |.| .|. .|. ._| ._| |_| |_|.  
//  |_| .|. |_| ._| .|. ._|
```

```
println(display(""))
```

```
//
```

```
//
```

```
//
```

# More Patterns

- Functional Programming in Scala (the red book)
- Scala with Cats book
- Patterns built on top of these abstractions
  - Free, Tagless Final, etc.

# Thank you!

Questions?

<https://github.com/richashworth/talks-public/tree/master/fp-patterns-in-scala>

Slides: <https://github.com/richashworth/talks-public/tree/master/fp-patterns-in-scala>