Updated: 2021-03-31

6.2
Lenses & Enumerated Types

Northeastern University

Lenses

Inverted *map*

- We all know that for a functor *F[A]*, the map method will be defined thus:
  - *def map[B](f: A=>B): F[B]*
- But, what if we were to have a method *x*:
  - *def x[B](f: B=>A): F[B]*
- What kind of a weird method is this *x*? Should we call it *unmap*? And how might we describe that function *f*? It's kind of backwards, right?

What is a lens?

What is a lens?

- A lens is essentially a functional way of accessing the fields of an object.
  - Normally, it's a mutating method but I don't think it has to be.
- Suppose that you have an *Employee* class:
  - `case Class Employee(name: String, salary: Int)`
  - You want the salary field to be private:
  - `case Class Employee(name: String, private val salary: Int)`
  - So, something like `employee.salary` won't compile because the *salary* field is private.
  - Nevertheless, some people like the employee's manager need to be able to see this field. So, we could generate a function of type `Employee => Int` that will extract the *salary* information from the given *Employee* record. You would only be able to generate this function if you had the appropriate credentials. Alternatively, a function `Int => Employee => Employee` would allow you to create a copy of an employee with a different salary (see next slide).
  - We would call this kind of function a *lens* function.

What is a lens (2)?

- Perhaps a more typical use of a lens function would be to copy an *Employee* but give the new copy a different *salary*.

```
      scala> val employee = Employee("Robin", 1000000)
      employee: Employee = Employee(Robin,1000000)

      scala> val updateSalary: Int=>Employee=>Employee = salary => employee =>
Employee(employee.name, salary)
      updateSalary: Int => (Employee => Employee) =
$$Lambda$953/1599488589@237b2852

      scala> updateSalary(1100000)(employee)
      res1: Employee = Employee(Robin,1100000)
```

Comparer

- I have an open-source project called *Comparer*:
  - if you did *lab-sorted* with me, you'll be familiar with the idea).
  - There is a method called *snap*:

```
def snap[U](f: U => T): Comparer[U] = u1 => u2 => self(f(u1))(f(u2))
```

  - It's like the *unmap* method we saw before.
  - The *U=>T* function *f* that it takes as a parameter is essentially a lens function: given a *U*, it will extract a *T*.

```
val ic = implicitly[Comparer[Int]]
val comparerY: Comparer[DateJ] = ic.snap(_.year)
val comparerM: Comparer[DateJ] = ic.snap(_.month)
val comparerD: Comparer[DateJ] = ic.snap(_.day)
val comparer: Comparer[DateJ] = comparerY orElse comparerM orElse comparerD
```

Enumerated Types

Enums

- Enums are a little more difficult in Scala than in Java (although Java enums certainly have plenty issues)
- There are essentially two ways to create enums:
  - case objects
  - extending *Enumeration*
- each has some advantages/disadvantages:
  - For detailed information see: StackOverflow

Cards by enumeration

```
object Rank extends Enumeration {
   type Rank = Value
   val Deuce, Trey, Four, Five, Six, Seven, Eight, Nine, Ten, Knave, Queen, King, Ace = Value
   class RankValue(rank: Value) {
     def isSpot = !isHonor
     def isHonor = rank match {
       case Ace | King | Queen | Knave | Ten => true
       case _ => false
     }
   }
   implicit def value2RankValue(rank: Value) = new RankValue(rank)
}
object Suit extends Enumeration {
   type Suit = Value
   val Clubs, Diamonds, Hearts, Spades = Value
   class SuitValue(suit: Value) {
      def isRed = !isBlack
      def isBlack = suit match {
         case Clubs | Spades => true
         case _              => false
      }
   }
   implicit def value2SuitValue(suit: Value) = new SuitValue(suit)
}
import Rank._
import Suit._
case class Card (rank: Rank, suit: Suit)
```

Cards by case object (1)

```scala
trait Concept extends Ordered[Concept]{
  val name: String
  val priority: Int
  override def toString = name
  def compare(that: Concept) = priority-that.priority
  def initial: String = name.substring(0,1)
}
sealed trait Rank extends Concept {
  def isHonor = priority > 7
  def isSpot = !isHonor
}
sealed trait Suit extends Concept {
  def isRed = !isBlack
  def isBlack = this match {
    case Spades | Clubs => true
    case _ => false
  }
}
object Concept {
  // This ordering gives the expected rank and suit (in bridge) order, at least for games where A
is considered to outrank K.
  implicit def ordering[A <: Concept]: Ordering[A] = Ordering.by(_.priority)
  // This ordering is for the traditional ordering for displaying bridge hands
  def reverseOrdering[A <: Concept]: Ordering[A] = ordering.reverse
}
```

Cards by case object (2)

```
case object Ace extends Rank {val name = "Ace"; val priority = 12 }
case object King extends Rank {val name = "King"; val priority = 11 }
case object Queen extends Rank {val name = "Queen"; val priority = 10 }
case object Knave extends Rank {val name = "Knave"; val priority = 9; override def initial = "J"}
case object Ten extends Rank {val name = "10"; val priority = 8; override def initial = "T"}
case object Nine extends Rank {val name = "9"; val priority = 7}
case object Eight extends Rank {val name = "8"; val priority = 6}
case object Seven extends Rank {val name = "7"; val priority = 5}
case object Six extends Rank {val name = "6"; val priority = 4}
case object Five extends Rank {val name = "5"; val priority = 3}
case object Four extends Rank {val name = "4"; val priority = 2}
case object Trey extends Rank {val name = "3"; val priority = 1}
case object Deuce extends Rank {val name = "2"; val priority = 0}
case object Spades extends Suit { val name = "Spades"; val priority = 3 }
case object Hearts extends Suit { val name = "Hearts"; val priority = 2 }
case object Diamonds extends Suit { val name = "Diamonds"; val priority = 1 }
case object Clubs extends Suit { val name = "Clubs"; val priority = 0 }
case class Card (suit: Suit, rank: Rank) extends Ordered[Card] {
  val bridgeStyle = true // as opposed to poker-style
  private def nameTuple = (suit.initial,rank.initial)
  override def toString = if (bridgeStyle) nameTuple.toString else nameTuple.swap.toString
  def compare(that: Card): Int = implicitly[Ordering[(Suit, Rank)]].compare(Card.unapply(this).get,
Card.unapply(that).get)
}
object Cards extends App {
  println(List(Card(Clubs,Deuce),Card(Clubs,King),Card(Clubs,Ten),Card(Spades,Deuce)).sorted)
}
```

More information

- The curious incident…
- Yuri’s blog
- Dotty enums