type classes for the masses

Javier Fuentes
Habla Computing
javier.fuentes@hablapps.com
@javifdev

Outline

- **What is a type class?**
- Type classes in Scala
- The type class pattern
- Type constructor classes

What is a type class?

- Let's write a function that takes a list of integers and adds them up.

```scala
def sum(numbers: List[Int]): Int =
  numbers match {
    case Nil => 0
    case x :: xs => x + sum(xs)
  }
```

What is a type class?

- As good programmers we are, we notice the recurring pattern.

```
def collapse[A](elements: List[A])(
    empty: A, combine: (A, A) => A): A =
  elements match {
    case Nil => empty
    case x :: xs =>
      combine(x, collapse(xs)(empty, combine))
  }
```

What is a type class?

- Someone thinks on wrapping the arguments and...

A type class!!!

```scala
trait Monoid[A] {
  val empty: A
  def combine(a1: A, a2: A): A
}

def collapse[A](elements: List[A])(m: Monoid[A]): A =
  elements match {
    case Nil => m.empty
    case x :: xs => m.combine(x, collapse(xs)(m))
  }
```

TYPE CLASS DEFINITION

```scala
trait Monoid[A] {
  val empty: A
  def combine(a1: A, a2: A): A
}
```

(AD-HOC) POLYMORPHIC FUNCTION

```scala
def collapse[A](l: List[A])(
    m: Monoid[A]): A =
  l.fold(m.empty)(m.combine)
```

TYPE CLASS INSTANCE

```scala
val intMonoid = new Monoid[Int] {
  val empty: Int = 0
  def combine(i1: Int, i2: Int) =
    i1 + i2
}
```

DEPENDENCY INJECTION

```scala
val i: Int = collapse(l)(intMonoid)
```

Outline

- What is a type class?
- **Type classes in Scala**
- The type class pattern
- Type constructor classes

Implicits...

- Let's start using some scala black magic.

```scala
def collapse[A](l: List[A])(implicit m: Monoid[A]) =
  l.foldLeft(m.empty)(m.combine)

implicit val intMonoid = new Monoid[Int] {
  val empty = 0
  def combine(i1: Int, i2: Int) = i1 + i2
}

collapse(List(1, 2, 3))
```

… Context bounds…

- Let’s start using some scala black magic.

```scala
def collapse[A: Monoid](l: List[A]) = {
  val m = implicitly[Monoid[A]]
  l.foldLeft(m.empty)(m.combine)
}

collapse(List(1, 2, 3))
```

… and syntax!

- Let’s start using some scala black magic.

```scala
object syntax {
  def empty[A](implicit ev: Monoid[A]) = ev.empty

  implicit class MonoidOps[A](a: A)(implicit ev: Monoid[A]) {
    def |+|(other: A): A = ev.combine(a, other)
  }
}

import syntax._
def collapse[A: Monoid](l: List[A]) =
  l.foldLeft(empty[A])(_ |+| _)

collapse(List(1, 2, 3))
```

Outline

- What is a type class?
- Type classes in Scala
- **The type class pattern**
- Type constructor classes

The type class pattern

A type class is made up of 5 parts:

- Abstract interface
- Concrete interface
- Instances
- Syntax
- Laws

Abstract	Concrete	Instances	Syntax	Laws

```scala
trait Order[A] {
  def compare(a1: A, a2: A): Int
  // ...
}
```

Abstract | Concrete | Instances | Syntax | Laws

```scala
trait Order[A] {
  // ...
  def gt(a1: A, a2: A): Boolean = compare(a1, a2) > 0
  def lt(a1: A, a2: A): Boolean = compare(a1, a2) < 0
  def eq(a1: A, a2: A): Boolean = compare(a1, a2) == 0
  def gteq(a1: A, a2: A): Boolean = !lt(a1, a2)
  def lteq(a1: A, a2: A): Boolean = !gt(a1, a2)
  def greater(a1: A, a2: A): A =
    if (gteq(a1, a2)) a1
    else a2
}
```

Abstract	Concrete	Instances	Syntax	Laws

```
object Order {
  def apply[A](implicit ev: Order[A]) = ev
  implicit val intInstance = new Order[Int] {
    def compare(i1: Int, i2: Int): Int = i1-i2
  }
  implicit val stringInstance: Order[String] = ???
  implicit def optionInstance[A](implicit ev: Order[A]) =
    new Order[Option[A]] {
      def compare(o1: Option[A], o2: Option[A]): Int =
        (o1, o2) match {
          case (Some(a1), Some(a2)) => ev.compare(a1, a2)
          case (None, None) => 0
          case (Some(_), _) => 1
          case _ => -1
        }
    }
  // ...
}
```

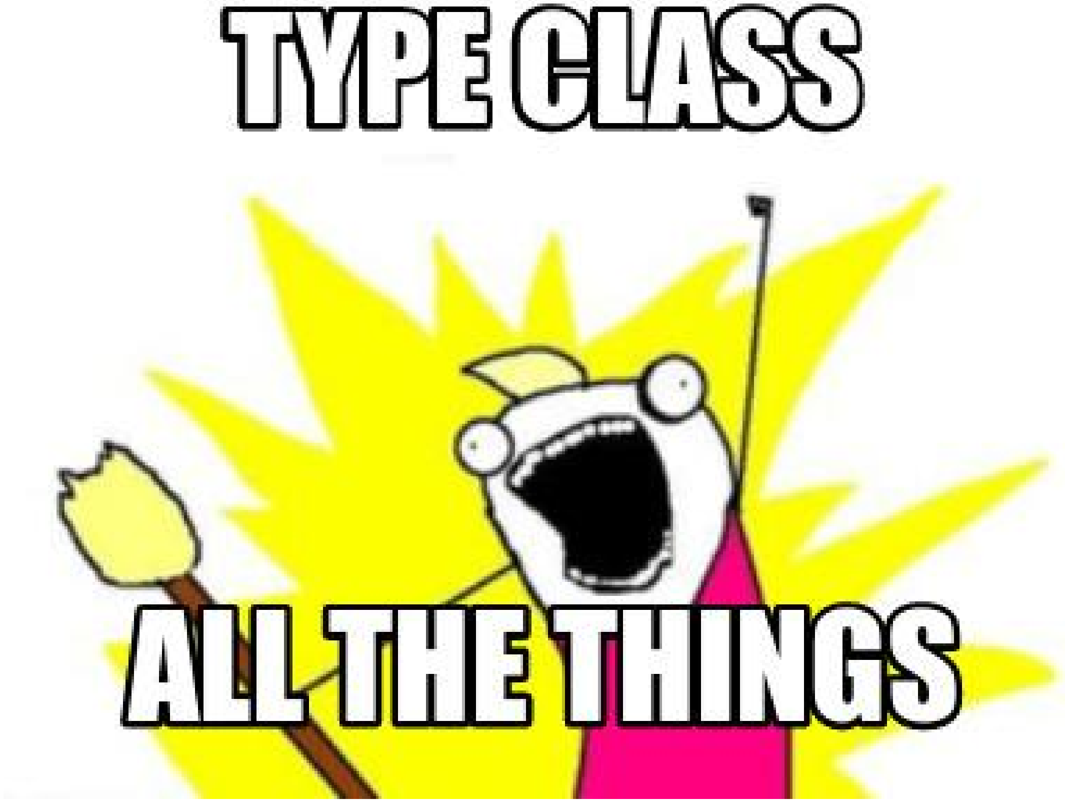
Abstract	Concrete	Instances	Syntax	Laws

```scala
object Order {
  // ...
  object syntax {
    implicit class OrderOps[A](a: A)(implicit ev: Order[A]) {
      def compareTo(other: A) = ev.compare(a, other)
      def >(other: A): Boolean = ev.gt(a, other)
      def <(other: A): Boolean = ev.lt(a, other)
      def ===(other: A): Boolean = ev.eq(a, other)
      def >=(other: A): Boolean = ev.gteq(a, other)
      def <=(other: A): Boolean = ev.lteq(a, other)
    }

    def greater[A](a1: A, a2: A)(implicit ev: Order[A]) =
      ev.greater(a1, a2)
  }
  // ...
}
```

Abstract | Concrete | Instances | Syntax | **Laws**

```scala
object Order {
  // ...
  trait OrderLaws[A: Order] {
    def antisymmetric(a1: A, a2: A): Boolean =
      (a1 > a2) == (a2 <= a1)

    def transitive(a1: A, a2: A, a3: A): Boolean = {
      val a1a2 = a1 > a2
      val a2a3 = a2 > a3
      val a1a3 = a1 > a3
      if (a1a2 == a2a3)
        a1a3 == a1a2
      else
        true
    }
  }
  // ...
}
```

We have everything we need so…



Let’s put this into practice

```scala
import Order.syntax._
def quicksortList[A: Order](l: List[A]): List[A] =
  l match {
    case a :: as =>
      quicksortList(as.filter(_ < a)) :::
      a ::
      quicksortList(as.filter(_ >= a))
    case Nil => Nil
  }

def maxList[A: Order](l: List[A]): Option[A] =
  l.foldLeft(Option.empty[A]) { (acc, a) =>
    acc.fold(Option(a)) { ac =>
      Option(greater(ac, a))
    }
  }
```


IT'S YOUR TURN
imgflip.com

Outline

- What is a type class?
- Type classes in Scala
- The type class pattern
- **Type constructor classes**

Type constructor classes

- Initial version of our program.

```scala
def echo: Unit = {
  val read = scala.io.StdIn.readLine
  println(read)
  // scala.io.StdIn.readLine andThen println
}
```

Type constructor classes

- Again, as good programmers we are, we "protect" ourselves using an interface.

```scala
trait IO {
  def read: String
  def write(msg: String): Unit
}
def echo(io: IO): Unit = {
  val read = io.read
  io.write(read)
}
val consoleIO = new IO {
  def read = scala.io.StdIn.readLine
  def write(msg: String) = println(msg)
}
echo(consoleIO)
```

Type constructor classes

- Let’s try to create an instance for an asyncronous platform...

```scala
val redisIO = new IO {
  import scala.concurrent.{Future, Await}
  // def read: Future[String] = ???
  def read: String = Await.result(???, ???)
  def write(msg: String): Unit = Await.result(???, ???)
}
```

INTERFACE

```scala
trait IO {
  def read: String
  def write(msg: String): Unit
}
```

PROGRAM OVER INTERFACE

```scala
def echo(io: IO): Unit = {
  val read = io.read
  io.write(read)
}
```

INTERFACE IMPLEMENTATION

```scala
val consoleIO = new IO {
  def read =
    scala.io.StdIn.readLine
  def write(msg: String) =
    println(msg)
}
```

INTERPRETATION

```scala
echo(consoleIO)
```

INTERFACE

```scala
trait IO {
  def read: Future[String]
  def write(msg: String)
    : Future[Unit]
}
```

PROGRAM OVER INTERFACE

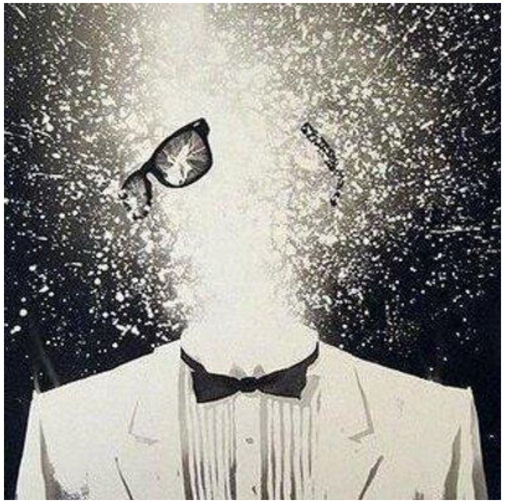
INTERFACE IMPLEMENTATION

INTERPRETATION

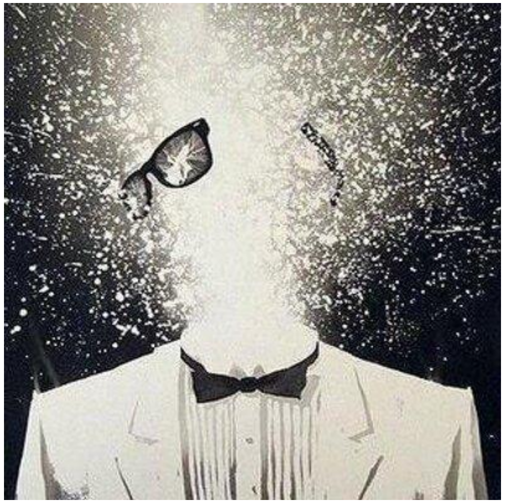
INTERFACE

```
trait IO {
  def read: State[S, String]
  def write(msg: String)
    : State[S, Unit]
}
```

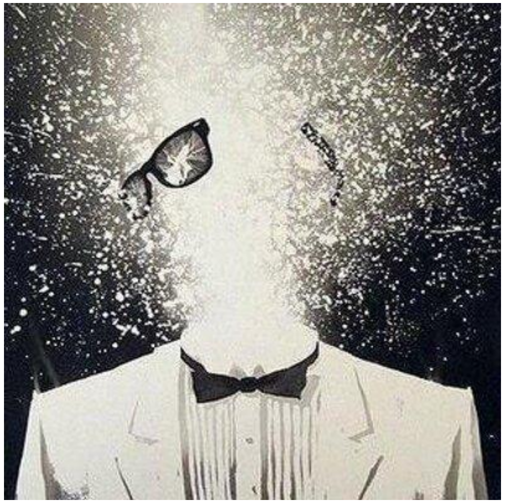
PROGRAM OVER INTERFACE



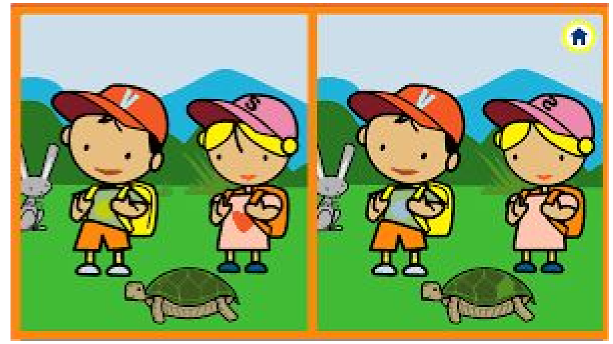
INTERFACE IMPLEMENTATION



INTERPRETATION



Find the seven differences



```scala
trait IO {
  def read: Either[Error, String]
  def write(msg: String): Either[Error, Unit]
}
```

```scala
trait IO {
  def read: State[S, String]
  def write(msg: String): State[S, Unit]
}
```

```scala
trait IO {
  def read: Future[String]
  def write(msg: String): Future[Unit]
}
```

```scala
trait IO {
  def read: Id[String]
  def write(msg: String): Id[Unit]
}
```

Type constructor classes

- Again, type class all the things!

```
trait IOAlg[F[_]] {
  def read: F[String]
  def write(msg: String): F[Unit]
}
object IOAlg {
  object syntax {
    def read[F[_]](implicit ev: IOAlg[F]) = ev.read
    def write[F[_]](msg: String)(implicit ev: IOAlg[F]) =
      ev.write(msg)
  }
}
```

Type constructor classes

- And... we failed!

```scala
import IOAlg.syntax._
def echo[F[_]: IOAlg]: F[Unit] = {
  val r: F[String] = read
  write(r)
}
```

Type constructor classes

- Monads to the rescue.

```scala
trait Monad[F[_]] {
  def flatMap[A, B](fa: F[A])(f: A => F[B]): F[B]
  def pure[A](a: A): F[A]
}
object Monad {
  object syntax {
    implicit class MonadOps[F[_], A](fa: F[A])(implicit ev: Monad[F]) {
      def flatMap[B](f: A => F[B]) = ev.flatMap(fa)(f)
      def map[B](f: A => B) = ev.flatMap(fa)(f andThen ev.pure)
    }
  }
}
```

Type constructor classes

- The final result.

```
import Monad.syntax._
def echo[F[_]: IOAlg: Monad]: F[Unit] = {
  for {
    r <- read
    _ <- write(r)
  } yield ()
  // read flatMap write[F]
}
```

Conclusion

- Type classes are a VERY flexible, VERY powerful technique in FP (and very underrated).
- They can be used for simple things as defining ordering operations and also for describing effects like IO.
- You can keep adding type class restrictions to your functions.

Hope you enjoyed it!