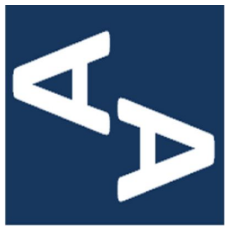
Programación funcional en Scala

2. Más allá de las HOFs

2.1 Type classes

2.2 Type classes vs. conventional OO

2.3 Type constructor classes



Habla Computing
info@hablapps.com
@hablapps

¿Por qué son importantes las type classes?

- Por su impacto en la **programación funcional**
- Patrón cercano a la programación **orientada a objetos**
- Código más **idiomático** en Scala que con ADTs
- Para conseguir mejorar la **modularidad** de nuestro código

Más allá de las HOFs

....

LENGUAJES

TYPE CLASSES

FUNCIONES DE ORDEN SUPERIOR

POLIMORFISMO PARAMÉTRICO

FUNCIONES

Objetivos

- Entender el papel de las *type classes* dentro del esquema de mecanismos de **modularidad**, y el soporte que ofrece Scala para este patrón de diseño

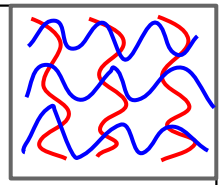
- Saber utilizar las type classes en situaciones donde utilizaríamos la herencia u otros patrones típicos de la **programación orientada a objetos**

Más allá de las HOFs

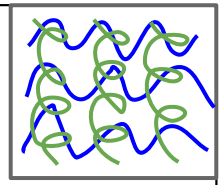
- **Sección 1: Type classes y la modularidad**
- Sección 2: Soporte de Scala
- Sección 3: Patrón de diseño
  - Ejercicio 1: Show
- Sección 4: Otros patrones OO
- Sección 5: Representación de datos
- Sección 6: Type constructor classes
  - Homework

Higher-order functions
(I) Programas monolíticos

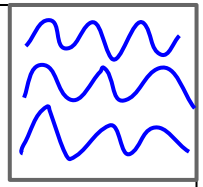
```scala
def sum(l: List[Int]): Int =
  l match {
    case Nil => 0
    case x :: r => x + sum(r)
  }
```



```scala
def concat(l: List[String]): String =
  l match {
    case Nil => ""
    case x :: r => x + concat(r)
  }
```



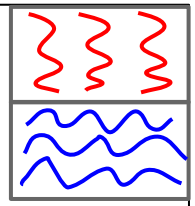
Higher-order functions
(II) Patrón recurrente



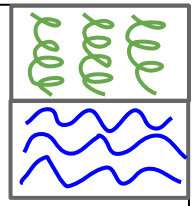
```scala
// Abstraemos los valores y funciones
def collapse[A](l: List[A])(zero: A, add: (A,A) => A): A =
  l match {
    case Nil => zero
    case x :: r => add(x,collapse(r)(zero,add))
  }
```

Higher-order functions
(III) Versiones modularizadas

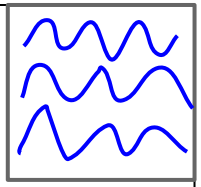
```
def sum(l: List[Int]): Int =
  collapse(l)(0, (i1,i2) => i1 + i2)
```



```
def concat(l: List[String]): String =
  collapse(l)(“”, _ + _)
```



Type classes
(II) Patrón recurrente

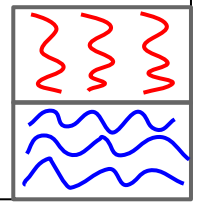


```scala
trait Monoide[T]{
  def add(t1: T, t2: T): T
  val zero: T
  // Más leyes: asociatividad y elemento neutro
}

def collapse[A](l: List[A])(monoid: Monoide[A]): A =
  l.fold(monoid.zero)(monoid.add)
```

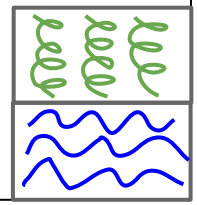
CON TYPE CLASSES

Type classes
(III) Versiones modularizadas

```
val intMonoid: Monoide[Int] = new Monoide[Int]{
  val zero: Int = 0
  def add(i1: Int, i2: Int): Int = i1 + i2
}

def sum(l: List[Int]): Int =
  collapse(l)(intMonoid)
```



```
object strMonoid extends Monoide[String]{
  val zero: String = ""
  def add(s1: String, s2: String): String = s1 + s2
}

def concat(l: List[String]): String =
  collapse(l)(strMonoid)
```



TYPE CLASS SIGNATURE / INTERFACE / FUNCTIONAL API

```
trait Monoide[T]{
  def append(t1: T, t2: T): T
  val zero: T
}
```

(AD-HOC) POLYMORPHIC FUNCTION / FUNCTIONS OVER API

```
def collapse[A](l: List[A])(
  monoid: Monoide[A]): A =
  l.fold(monoid.zero)(monoid.add)
```

TYPE CLASS INSTANCE / INTERPRETER INTERFACE IMPLEMENTATION

```
val intMonoid = new Monoide[Int] =
  val zero: Int = 0
  def add(i1: Int, i2: Int): Int =
    i1 + i2
}
```

INTERPRETATION / DEPENDENCY INJECTION

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

¿Qué son las *type classes*?

- Una type class es un ***diccionario*** de funciones y valores indexado por **tipos** de datos

```
trait Show[T] {
  def toString(t: T): String
}
```

```
trait Serializable[T] {
  def toBytes(t: T): Array[Byte]
  def fromBytes(s: Array[Byte]): T
}
```

```
trait Equal[T] {
  def equal(t1: T, t2: T): Boolean
}
```

```
trait Ordering[T] {
  def compare(t1: T, t2: T): Int
}
```

¿Qué son las *type classes*?

- Las *type classes* son interfaces genéricas que definen una funcionalidad que proporciona el tipo que parametrizan (métodos, valores, ...)
  - La implementación de esta funcionalidad (métodos, valores, etc.) puede estar sujeta a **leyes**
  - E.g.: Monoides (asociatividad e identidad)
- ¿Por qué? Capturan funcionalidad altamente **reutilizable** de una manera **extensible**, facilitando la **corrección**

Características deseables:
Expresividad y Generalidad

- Generalidad
  - Una *type class* debe poder clasificar muchos tipos
  - Ejemplo: hay muchísimas instancias de *monoides*, es decir, de tipos cuyos valores nos gustaría combinar
- Expresividad
  - Número de operaciones derivadas que podré definir a partir de las operaciones primitivas
  - Ejemplo monoides: multiply, ifEmpty, onEmpty, ...

```
EXPRESIVIDAD y GENERALIDAD... ¿son complementarios o se
contradicen entre sí?
```

Expresividad y generalidad

```
trait Ordering[T] {
  // Primitive operations
  def compare(x: T, y: T): Int

  // Derived operations
  override def lteq(x: T, y: T): Boolean = ...
  override def gteq(x: T, y: T): Boolean = ...
  override def lt(x: T, y: T): Boolean = ...
  def max(x: T, y: T): T = ...
  def min(x: T, y: T): T = ...
  override def reverse: Ordering[T] = new Ordering[T] {
    override def reverse = outer
    def compare(x: T, y: T) = outer.compare(y, x)
  }
  ...
}
```

EXPRESIVIDAD

Expresividad y generalidad

```
object Ordering{
  object UnitOrdering extends Ordering[Unit] {
    def compare(x: Unit, y: Unit) = 0
  }

  object BooleanOrdering extends Ordering[Boolean] {
    def compare(x: Boolean, y: Boolean) = (x, y) match {
      case (false, true) => -1
      case (true, false) => 1
      case _ => 0
    }
  }

  object CharOrdering extends Ordering[Char]
  object IntOrdering extends Ordering[Int]
  object StringOrdering extends Ordering[String]
  ...

}
```

GENERALIDAD

Diseño de *type class* & implementación

Interfaz	Operaciones primitivas que deben implementarse para los tipos de la type class
Leyes	Propiedades que deben satisfacer dichas operaciones
Operaciones	Operaciones derivadas definidas en términos de las operaciones primitivas u otras operaciones derivadas; cuantas más operaciones derivadas tenga una type class, más **expresiva** será
Instancias	Instanciaciones de esta clase para diferentes tipos; cuantas más instancias se pueden dar de una type class, más **general** será

Más allá de las HOFs

- Sección 1: Type classes y la modularidad
- **Sección 2: Soporte de Scala**
- Sección 3: Patrón de diseño
  - Ejercicio 1: Show
- Sección 4: Otros patrones OO
- Sección 5: Representación de datos
- Sección 6: Type constructor classes
  - Homework

Type classes en Scala
Implicits + context bounds

- Objetivo: enseñar el código idiomático en Scala para *type classes*
  - Aún tengo que pasar la instancia de **`Monoide`** a la función ¿Puedo evitarlo?

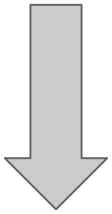
```scala
// (II) Patrón recurrente
def collapse[A](l: List[A])(monoid: Monoide[A]): A =
  l.fold(monoid.zero)(monoid.add)

// (III) Versión modularizada
def sum(l: List[Int]): Int = collapse(l)(intMonoid)
def concat(l:List[String]): String = collapse(l)(strMonoid)
```

Type classes con *implicits*

```scala
// (II) Patrón recurrente (implícitos)
def collapse[A](l:List[A])(implicit monoid: Monoide[A]): A =
  l.foldLeft(monoid.zero)(monoid.add)

// (III) Versión modularizada
implicit val intMonoid: Monoide[Int] = new Monoide[Int]{
  val zero = 0
  def add(i1: Int, i2: Int): Int = i1 + i2
}
implicit object stringMonoid extends Monoide[String]{
  val zero: String = ""
  def add(s1: String, s2: String): String = s1 + s2
}

def sumaInt(l: List[Int]): Int = collapse(l)
def concat(l: List[String]): String = collapse(l)
```

Inferencia de implícitos

```
val b1: Boolean =
  (Option(2),"hola") lteq (Option(2), "iii")
```



```
val b2: Boolean =
  (new Ops((Option(2),"hola"))(
    Ordering.Tuple2[Option[Int],String](
      Ordering.Option[Int](Ordering.Int),
      Ordering.String)
    )
  ).lteq((Option(2), "iii"))
```

Type classes con *context bounds*

```scala
// (II) Patrón recurrente (context bounds, con implicitly)
def collapse[A: Monoide](l:List[A]): A = {
  val monoid = implicitly[Monoide[A]]
  l.foldLeft(monoid.zero)(monoid.add)
}
```

```scala
//(II) Patrón recurrente (context bounds, sin implicitly)

import MonoidSyntax._
def collapse[A: Monoide](l:List[A]): A =
  l.foldLeft(zero)(_ add _)
```

Type classes - *context bound*

```
def f[T: TypeClass](t: T)
// T:TypeClass → “T es una “instancia” de TypeClass”
// t:T         → “t es una instancia de T”
```

`[T: TypeClass]` expresa que el tipo `T` pertenece a `TypeClass`, y, por tanto, que sus métodos y valores están disponibles en la función *f*

Polimorfismo paramétrico vs. *ad-hoc*

- Polimorfismo paramétrico
  - El código está parametrizado con respecto a un tipo T, *del que no sabemos nada*
- Polimorfismo con type classes, o ad-hoc
  - El código no es solo paramétrico en **T**: las funciones reciben info extra (*ad-hoc*) sobre **T**
  - Una *type class* puede dar información extra para convertir, serializar, combinar... valores de tipo **T**
    - `Show[T]`, `Equals[T]`, `Ordering[T]`, `Monoid[T]`, `...`

Más allá de las HOFs

- Sección 1: Type classes y la modularidad
- Sección 2: Soporte de Scala
- **Sección 3: Patrón de diseño**
  - Ejercicio 1: Show
- Sección 4: Otros patrones OO
- Sección 5: Representación de datos
- Sección 6: Type constructor classes
  - Homework

Abstract	Concrete	Instances	Syntax	Laws

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

  // ...

}
```

```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
}
```

```scala
trait OrderInstances {
  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
        }
    }
}
```

```scala
trait OrderSyntax {
  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

```
trait OrderLaws {
  import Order.syntax._

  trait Laws[A] {
    implicit val instance: Order[A]
    ...
    def antisymmetric(a1: A, a2: A): Boolean =
      (a1 > a2) == (a2 <= a1)
  }

  object Laws {
    def apply[A](implicit ev: Order[A]) =
      new Laws[A] {
        implicit val instance: Order[A] = ev
      }
  }
}
```

Abstract | Concrete | Instances | Syntax | Laws


```scala
trait Order[A] {
  // 1. Abstract interface
  /* ... */

  // 2. Concrete interface
  /* ... */
}

object Order extends OrderInstances
  with OrderSyntax
  with OrderLaws

// 3. Instances (including caster)
trait OrderInstances {/* ... */}
// 4. Syntax
trait OrderSyntax {/* ... */}
// 5. Laws
trait OrderLaws {/* ... */}
```

Más allá de las HOFs

- Sección 1: Type classes y la modularidad
- Sección 2: Soporte de Scala
- Sección 3: Patrón de diseño
  - **Ejercicio 1: Show**
- Sección 4: Otros patrones OO
- Sección 5: Representación de datos
- Sección 6: Type constructor classes
  - Homework

Ejercicios type classes

Implementar la type class `Show[A]` que define la clase de los tipos que pueden ser representados mediante un `String`
en **tema2-typeclasses/exercise1/TypeClasses.scala**

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

Más allá de las HOFs

- Sección 1: Type classes y la modularidad
- Sección 2: Soporte de Scala
- Sección 3: Patrón de diseño
  - Ejercicio 1: Show
- **Sección 4: Otros patrones OO**
- Sección 5: Representación de datos
- Sección 6: Type constructor classes
  - Homework

Herencia … ¿Por qué no?

```
abstract class Any {
  def equals(that: Any): Boolean
  def hashCode(): Int
  def toString(): String
  ...
}
```


```
case class Tuple2[+T1,+T2](
  _1: T1, _2: T2) ...{
  override def toString() =
    "(" + _1 + "," + _2 + ")"
  ...
}
```

```
trait Function1[-T1, +R]
  extends AnyRef {
  ...
  override def toString() =
    "<function1>"
}
```

Herencia … ¿Por qué no?

- Hay veces que no tiene sentido heredar esa funcionalidad para una tipo dado
- Hay veces que para un mismo tipo es posible que queramos distintas implementaciones de esa funcionalidad
- Hay veces que no podemos anticipar toda la funcionalidad que vamos a querer
- La herencia se lleva fatal con la inmutabilidad

Adaptadores vs. type classes

```
abstract class Ordering[T](val unwrap: T){
  def compare(other: T): Int
  def gteq(t2: T): Boolean = ...
  def eq(t2: T): Boolean = ...
  def lteq(t2: T): Boolean = ...
}
```

```
trait Ordering[T] {
  def compare(t1: T, t2: T): Int
  def gteq(t1:T, t2:T): Boolean = ...
  def eq(t1:T, t2:T): Boolean = ...
  def lteq(t1:T, t2:T): Boolean = ...
}
```

Adaptadores vs. type classes

```scala
def greatest[A](l: List[A])(
  wrap: A => Ordering[A]): Option[A] =
  l.sortWith(wrap(_) greaterThan _)
   .headOption
```

```scala
def greatest[A](l: List[A])(
  implicit ord: Ordering[A]): Option[A] =
  l.sortWith(ord.gteq)
   .headOption
```

Adaptadores … ¿Por qué no?

- Los adaptadores son más ineficientes: hay que crear tantas instancias del adaptador como *objetos* adaptados
  - Con type classes, hay que crear solo una instancia por *tipo*
- En ocasiones no es posible siquiera utilizarlos
  - Por ejemplo, cuando la información es estática

Adaptadores ... ¿Por qué no?

```scala
trait Monoide[T]{
  val zero: T
  def add(t1: T,t2: T): T
}
```

```scala
abstract class Monoide1[T](val unwrap: T){
  val zero: T
  def add(t2: T): T
}
```

¿¿¿¡¡¡Un zero distinto por instancia!!!???

Más allá de las HOFs

- Sección 1: Type classes y la modularidad
- Sección 2: Soporte de Scala
- Sección 3: Patrón de diseño
  - Ejercicio 1: Show
- Sección 4: Otros patrones OO
- **Sección 5: Representación de datos**
- Sección 6: Type constructor classes
  - Homework

Representación de datos con type classes

- Las type classes no solo se pueden utilizar para representar funcionalidad genérica que queramos *añadir* a un tipo existente (o por venir)
  - Ej. La clase de los tipos que ***se pueden*** comparar
- También pueden utilizarse para representar los propios tipos de datos
  - La funcionalidad que proporciona la type class son los propios *constructores* del tipo de datos
  - Ej., la clase de los tipos que ***son*** expresiones arit.

Tipos de datos como type classes
Similar a las factorías abstractas

```scala
trait Exp[E] {
  def lit(i: Int): E
  def add(e1: E, e2: E): E
}
```

TCs

```scala
sealed trait ADTExp
case class Lit(x: Int) extends ADTExp
case class Add(l: Exp, r: Exp) extends ADTExp
```

ADTs

```scala
// Creación de ADT `Expr` mediante la type class
object ADTExp extends Exp[ADTExp]{
  def lit(i: Int): ADTExp = Lit(i)
  def add(e1: ADTExp, e2: ADTExp): ADTExp = Add(e1,e2)
}
```

TYPE CLASS SIGNATURE / INTERFACE (API)

```
trait Exp[E] {
  def lit(i: Int): E
  def add(e1: E, e2: E): E
}
```

(AD-HOC) POLYMORPHIC FUNCTION / FUNCTIONS OVER API

```
def op[E](E: Exp[E]): E =
  E.add(E.lit(1), E.lit(2))
```

TYPE CLASS INSTANCE / INTERPRETER INTERFACE IMPLEMENTATION

```
object ADTExp extends Exp[ADTExp]{
  def lit(i: Int): ADTExp = Lit(i)
  def add(e1: ADTExp, e2: ADTExp)=
    Add(e1,e2)
}
```

INTERPRETATION / DEPENDENCY INJECTION

```
val i: ADTExp = op(ADTExp)
```

Type classes vs. ADTs
Funcionalidad

ADTs

```
def eval(e: Exp): Int = e match {
  case Lit(i) => i
  case Add(l, r) => eval(l) + eval(r)
}
```

TCs

```
object eval extends Exp[Int] {
  def lit(i: Int): Int = i
  def add(e1: Int, e2: Int): Int =
    e1 + e2
}
```

TYPE CLASS SIGNATURE / INTERFACE / FUNCTIONAL API

```
trait Exp[E] {
  def lit(i: Int): E
  def add(e1: E, e2: E): E
}
```

(AD-HOC) POLYMORPHIC FUNCTION / FUNCTIONS OVER API

```
def op[E](E: Exp[E]): E =
  E.add(E.lit(1), E.lit(2))
```

TYPE CLASS INSTANCE / INTERPRETER INTERFACE IMPLEMENTATION

```
object ADTExp extends Exp[ADTExp]{
  def lit(i: Int): ADTExp = Lit(i)
  def add(e1: ADTExp, e2: ADTExp)=
    Add(e1,e2)
}
```

```
object eval extends Exp[Int] {
  def lit(i: Int): Int = i
  def add(e1: Int, e2: Int)=
    e1 + e2
}
```

INTERPRETATION / DEPENDENCY INJECTION

```
val e: ADTExp = op(ADTExp)

/* indirect evaluation */
val i: Int = adt.eval(op(ADTExp))
```

```
/* direct evaluation! */
val v: Int = op(eval)
```

Más allá de las HOFs

- Sección 1: Type classes y la modularidad
- Sección 2: Soporte de Scala
- Sección 3: Patrón de diseño
  - Ejercicio 1: Show
- Sección 4: Otros patrones OO
- Sección 5: Representación de datos
- **Sección 6: Type constructor classes**
  - Homework

Higher-kinds generics

```
// T (*)
String
Int
Potato
Option[Potato] // type T = Option[Potato]
Either[String, Int] // type T = Either[String, Int]

// T[_] (* -> *)
List[?]
Option[?]
Either[String, ?] // type T[X] = Either[String, X]

// T[_, _] (* -> * -> *)
Either[?, ?]
```

Type-constructor classes

```scala
def mapList[A, B](l: List[A])(f: A => B): List[B] =
  l.foldRight(List.empty[B])((a, acc) => f(a) :: acc)

def mapOpt[A, B](o: Option[A])(f: A => B): Option[B] =
  o.fold(Option.empty[B])(f andThen Option.apply)

def duplicateList[A](l: List[A]): List[(A, A)] =
  mapList(l)(a => (a, a))

def duplicateOpt[A](o: Option[A]): Option[(A, A)] =
  mapOpt(o)(a => (a, a))
```

Type-constructor classes

```
def filter(l: List[Int   ])(cond: Int    => Boolean): List[Int   ]
def filter(l: List[String])(cond: String => Boolean): List[String]

// (*)
Int      \
String    >==========> A
Boolean  /

def map[A, B](l: List  [A])(f: A => B): List  [B]
def map[A, B](o: Option[A])(f: A => B): Option[B]

// (* -> *)
List[_]   \
           >==========> F[_]
Option[_] /
```

Type-constructor classes

```scala
trait Functor[F[_]]{
  // 1. Abstract
  def map[A, B](fa: F[A])(f: A => B): F[B]

  // 2. Concrete
  def lift[A, B](f: A => B): F[A] => F[B] = map(_)(f)
  def as[A, B](fa: F[A], b: B): F[B] = map(fa)(_ => b)
}
```

Type-constructor classes

```scala
// 3. Instances
trait FunctorInstances {
  def apply[F[_]](implicit ev: Functor[F]) = ev

  implicit val optionFunctor = new Functor[Option]{
    def map[A, B](fa: Option[A])(f: A => B): Option[B] =
      fa map f
  }
}
```

Type-constructor classes

```scala
// Versiones monolíticas
def duplicateList[A](l: List[A]): List[(A, A)] =
  map(l)(a => (a, a))

def duplicateOpt[A](o: Option[A]): Option[(A, A)] =
  map(o)(a => (a, a))

// Version genérica
import Functor.syntax._
def duplicate[F[_]: Functor, A](o: F[A]): F[(A, A)] =
  o.map(a => (a, a))
```

Más allá de las HOFs

- Sección 1: Type classes y la modularidad
- Sección 2: Soporte de Scala
- Sección 3: Patrón de diseño
  - Ejercicio 1: Show
- Sección 4: Otros patrones OO
- Sección 5: Representación de datos
- Sección 6: Type constructor classes
  - **Homework**

Ejercicios para casa

- **`Ej.1  tema2-.../homework/EjercicioTypeClasses.scala`**
  - Impl. *type class* para cálculos estadísticos
  - Comparar con una solución monolítica
- **`Ej.2  tema2-.../homework/EjercicioTypeConstructors.scala`**
  - Se proporciona una *type class* para trabajar con colecciones de enteros
  - Se pide crear una *type class* para trabajar con colecciones de elementos cualesquiera
  - Para ello se deberá crear una *type class* con genericidad *higher-kind*

Conclusiones

- Las type classes son uno de los patrones de diseño funcional más potente
  - Permiten definir APIs mucho más modulares (extensibles, reutilizables)
  - Relacionados con los adaptadores, factorías, visitors, ...
- En scala tienen muy buen soporte
  - Implícitos, implicit classes, traits, higher-kind generics, ...
- *Type constructor classes*: más allá de lo que se puede hacer en Java
- Utiliza las type classes no solo para representar funcionalidad genérica, sino también los propios tipos de datos de manera abstracta
  - Busca la generalidad y la expresividad