



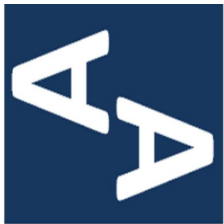
Programación funcional en Scala

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](https://twitter.com/hablapps)

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**
- Resolver problemas de **extensibilidad** mediante la declaración de tipos de datos con *type classes*

Más allá de las HOFs

....

LENGUAJES

TYPE CLASSES

FUNCIONES DE ORDEN SUPERIOR

POLIMORFISMO PARAMÉTRICO

FUNCIONES



Más allá de las HOFs

- ❖ **Sección 1: type classes y la modularidad**
- ❖ Sección 2: Implicits y context bounds
- ❖ Ejercicio 1: Comparable
- ❖ Sección 3: Type classes vs. herencia/adaptadores
- ❖ Sección 4: Type classes vs. factorías abstractas
- ❖ Sección 5: Type classes vs. visitors
- ❖ Sección 6: Extensibilidad de datos y funcionalidad
- ❖ Sección 7: Type constructor classes
- ❖ Homework

¿Qué son las *type classes*?

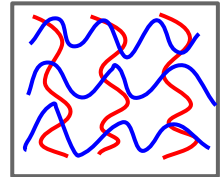
- No confundir con las *clases* de la programación orientada a objetos
 - Hablamos de *clases de tipos*, no de *clases de objetos*
- Las *type classes*
 - Permiten *clasificar* tipos en base a una serie de operaciones o valores
 - Son un mecanismo de modularidad: nos permiten extender, modificar, reutilizar, etc., nuestro código más fácilmente

Type classes

(I) Programas monolíticos

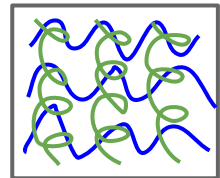
// Ejemplo 1

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



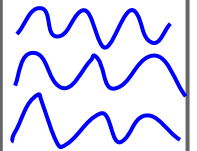
// Ejemplo 2

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



Type classes

(II) Patrón recurrente



// Abstraemos los valores y funciones

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

SIN TYPE CLASSES

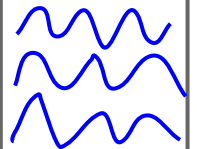
Type classes

(II) Patrón recurrente

```
trait Monoid[T] {                                     // Más leyes:
  def add(t1: T, t2: T): T // - Asociatividad
  val zero: T              // - Elemento neutro
}
```

// Abstraemos la type class Monoid

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



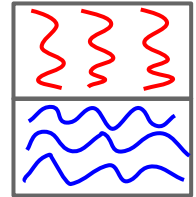
CON TYPE CLASSES

Type classes

(III) Versiones modularizadas

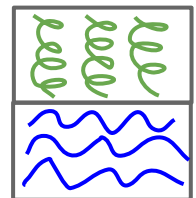
// Ejemplo 1

```
val intMonoid: Monoid[Int] = new Monoid[Int] {  
  val zero: Int = 0  
  def add(i1: Int, i2: Int): Int = i1 + i2  
}  
def sum(l: List[Int]): Int = collapse(l)(intMonoid)
```



// Ejemplo 2

```
object strMonoid extends Monoid[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 Monoid[A]:  
  def append(t1: A, t2: A): A  
  val zero: A  
}
```

(AD-HOC) POLYMORPHIC FUNCTION /
FUNCTIONS OVER API

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

TYPE CLASS INSTANCE / INTERPRETER
INTERFACE IMPLEMENTATION

```
val intMonoid = new Monoid[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*?

- 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 **modular**, 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?



Más allá de las HOFs

- ❖ Sección 1: type classes y la modularidad
- ❖ **Sección 2: Implicits y context bounds**
- ❖ Ejercicio 1: Comparable
- ❖ Sección 3: Type classes vs. herencia/adaptadores
- ❖ Sección 4: Type classes vs. factorías abstractas
- ❖ Sección 5: Type classes vs. visitors
- ❖ Sección 6: Extensibilidad de datos y funcionalidad
- ❖ Sección 7: 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?

```
// (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 *implicit*s

```
// (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)
```

Type classes con *context bounds*

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

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

import MonoidSyntax._
def collapse[A: Monoid](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: Implicits y context bounds
- ❖ **Ejercicio 1: Comparable**
- ❖ Sección 3: Type classes vs. herencia/adaptadores
- ❖ Sección 4: Type classes vs. factorías abstractas
- ❖ Sección 5: Type classes vs. visitors
- ❖ Sección 6: Extensibilidad de datos y funcionalidad
- ❖ Sección 7: Type constructor classes
- ❖ Homework

Ejercicios type classes

Modularizar programas mediante type classes
en **tema2-typeclasses/EjerciciosClase_TypeClasses.scala**

```
def sortAscendingChar(l: List[Char]): List[Char] =  
  l.sortWith( (c1: Char, c2: Char) => c1 < c2 )  
  
def sortAscendingInt(l: List[Int]): List[Int] =  
  l.sortWith( (c1: Int, c2: Int) => c1 < c2 )  
  
def sortAscendingString(l: List[String]): List[String] =  
  l.sortWith( (c1: String, c2: String) => c1 < c2 )
```



```
trait Comparable[T] {  
  // Operaciones primitivas  
  def compare(t1: T, t2: T): Int  
  // Operaciones derivadas  
  def greaterThan(t1: T, t2: T): Boolean = ???  
  def equalThan(t1: T, t2: T): Boolean = ???  
  def lowerThan(t1: T, t2: T) = ???  
}
```

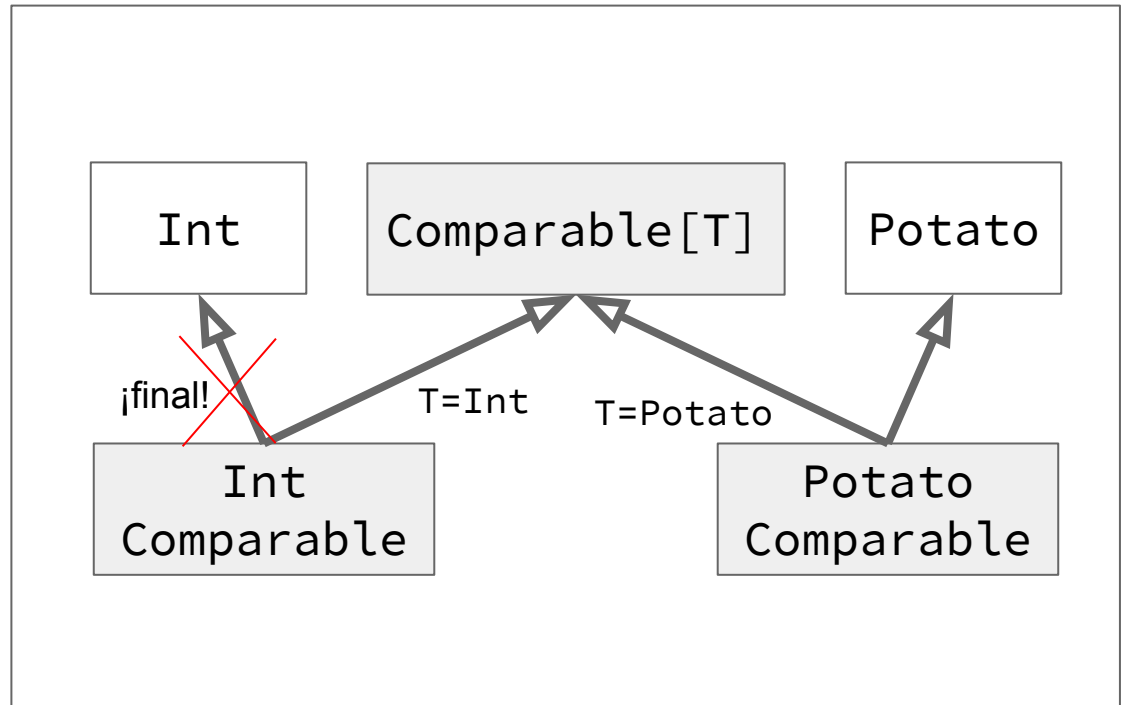
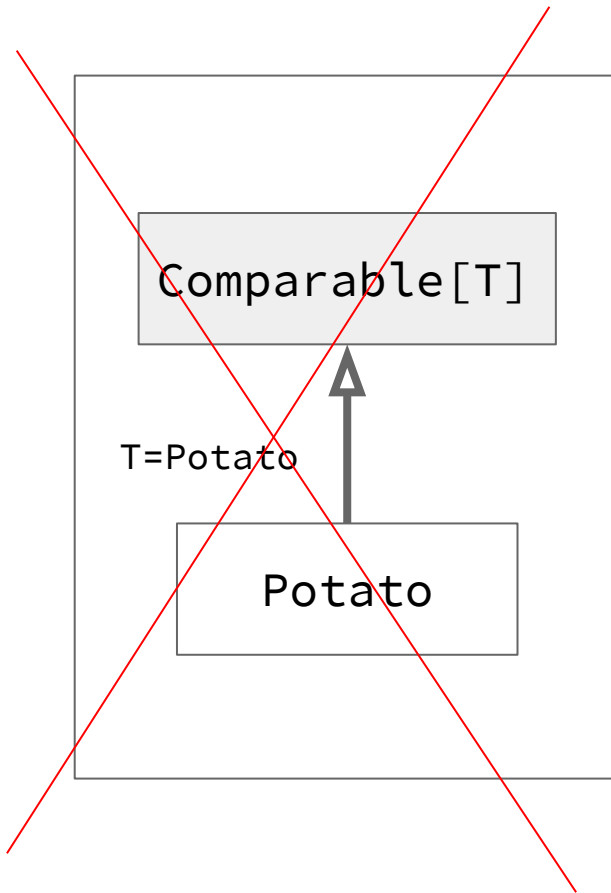


Más allá de las HOFs

- ❖ Sección 1: type classes y la modularidad
- ❖ Sección 2: Implicits y context bounds
- ❖ Ejercicio 1: Comparable
- ❖ **Sección 3: Type classes vs. herencia/adaptadores**
- ❖ Sección 4: Type classes vs. factorías abstractas
- ❖ Sección 5: Type classes vs. visitors
- ❖ Sección 6: Extensibilidad de datos y funcionalidad
- ❖ Sección 7: Type constructor classes
- ❖ Homework

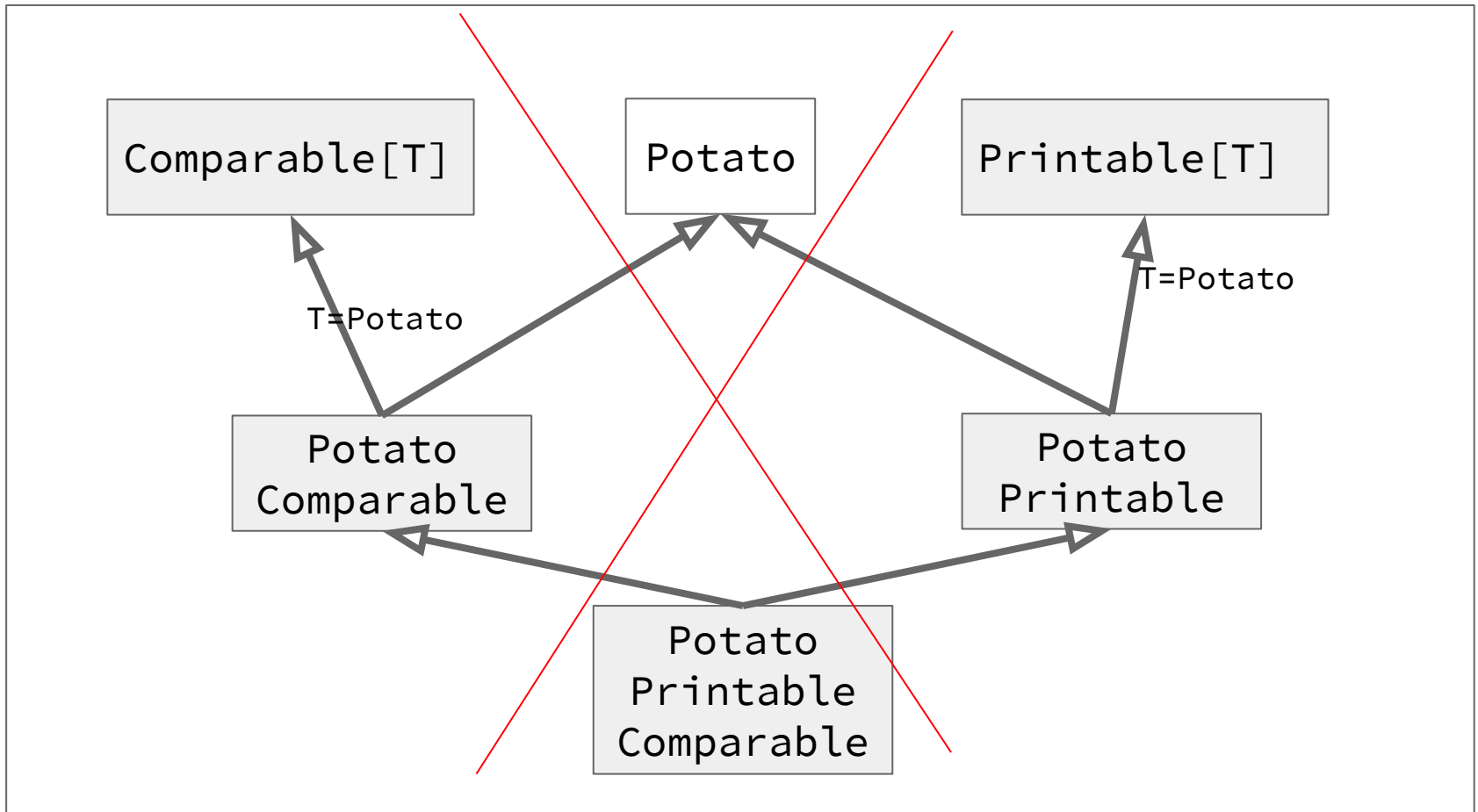
Type classes

¿Y por qué no la *herencia*?



Type classes

¿Y por qué no la *herencia*?

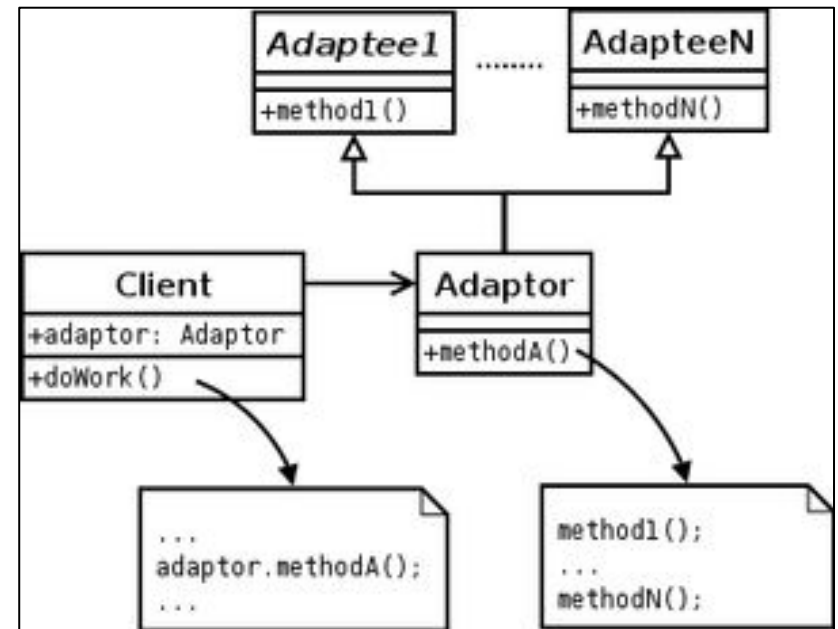
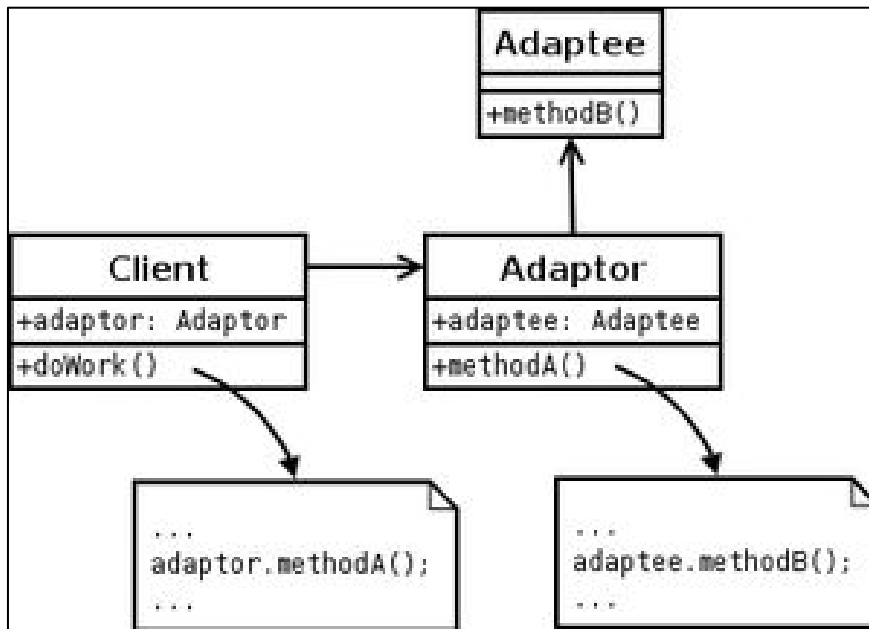


¿Por qué no utilizar herencia?

- Porque es posible que las clases que queramos extender ya existan
 - Pero podríamos crear nuevas subclases suyas
- Porque es posible que las clases que queramos extender sean *finales*
 - En ese caso no podríamos crear subclases
- Porque aunque podamos crear subclases, tendríamos que crear nuevas subclases para combinar otras posibles extensiones

El patrón *Adaptador*

ESTA VERSIÓN NO ES VIABLE
SI LAS CLASES A ADAPTAR
SON **FINALES**



https://en.wikipedia.org/wiki/Adapter_pattern

Adaptor vs. type classes

```
trait Comparable[A] {  
  def compare(t1: A, t2: A): Int  
  // derived  
  def gt(t1: A, t2: A): Boolean = compare(t1, t2) > 0  
  def eq(t1: A, t2: A): Boolean = compare(t1, t2) == 0  
  def lt(t1: A, t2: A): Boolean = compare(t1, t2) < 0  
}
```

```
trait Comparable[A] {  
  val unwrap: A  
  def compare(t2: A): Int  
  // derived  
  def gt(t2: A): Boolean = compare(t2) > 0  
  def eq(t2: A): Boolean = compare(t2) == 0  
  def lt(t2: A): Boolean = compare(t2) < 0  
}
```

Adaptor vs. type classes

Patrón recurrente

```
def sortAscending[A](l: List[A])(  
  wrap: A => Comparable[A]): List[A] =  
  l.map(wrap)  
    .sortWith((c1, c2) => c1.lowerThan(c2.unwrap))  
    .map(_.unwrap)
```

```
def sortAscending[A](l: List[A])(  
  implicit C: Comparable[A]): List[A] =  
  l.sortWith(C.lowerThan)
```

Adaptadores

¿Por qué no?


- Porque hay que crear tantas instancias del adaptador como *objetos* adaptados
 - Con type classes, hay que crear solo una instancia por *tipo*
- Porque hay que *wrapear y deswrapear* constantemente los objetos
 - Con type classes, no hay ninguna necesidad
- Porque en ocasiones no es posible siquiera utilizarlos
 - Por ejemplo, cuando la información es estática

¿Monoides como adaptadores?

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

```
trait Monoid1[T] {  
  val unwrap: T  
  val zero: T  
  def add(t2: T): T  
}
```

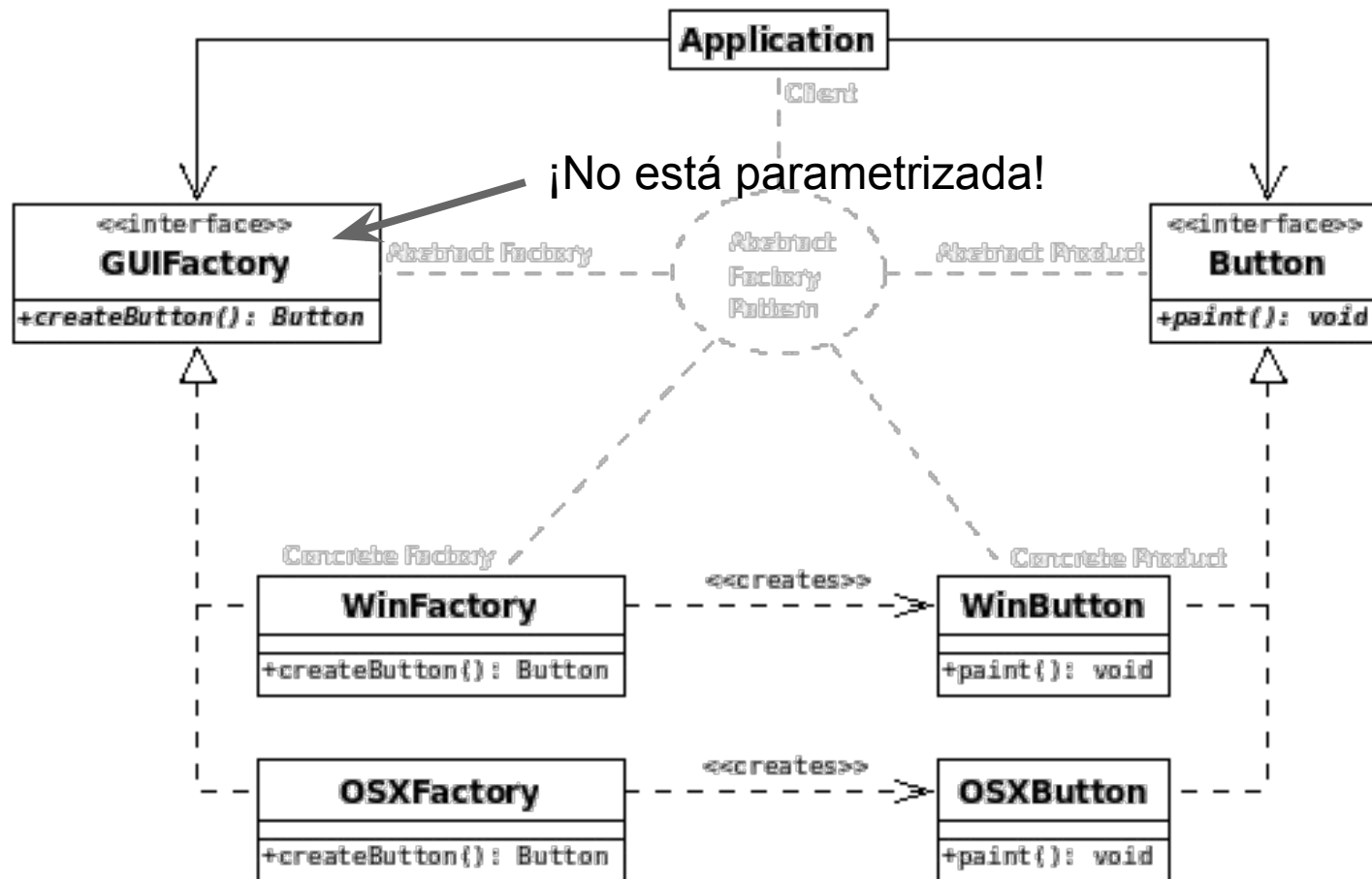
¿¿¿!!! Un zero distinto
por instancia!!!???



Más allá de las HOFs

- ❖ Sección 1: type classes y la modularidad
- ❖ Sección 2: Implicits y context bounds
- ❖ Ejercicio 1: Comparable
- ❖ Sección 3: Type classes vs. herencia/adaptadores
- ❖ **Sección 4: Type classes vs. factorías abstractas**
- ❖ Sección 5: Type classes vs. visitors
- ❖ Sección 6: Extensibilidad de datos y funcionalidad
- ❖ Sección 7: Type constructor classes
- ❖ Homework

Abstract factory pattern



Representación de datos con type classes

- Las type classes no solo se pueden utilizar para representar funcionalidad genérica que queremos *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

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

TCs

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

ADTs

```
// 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)  
}
```

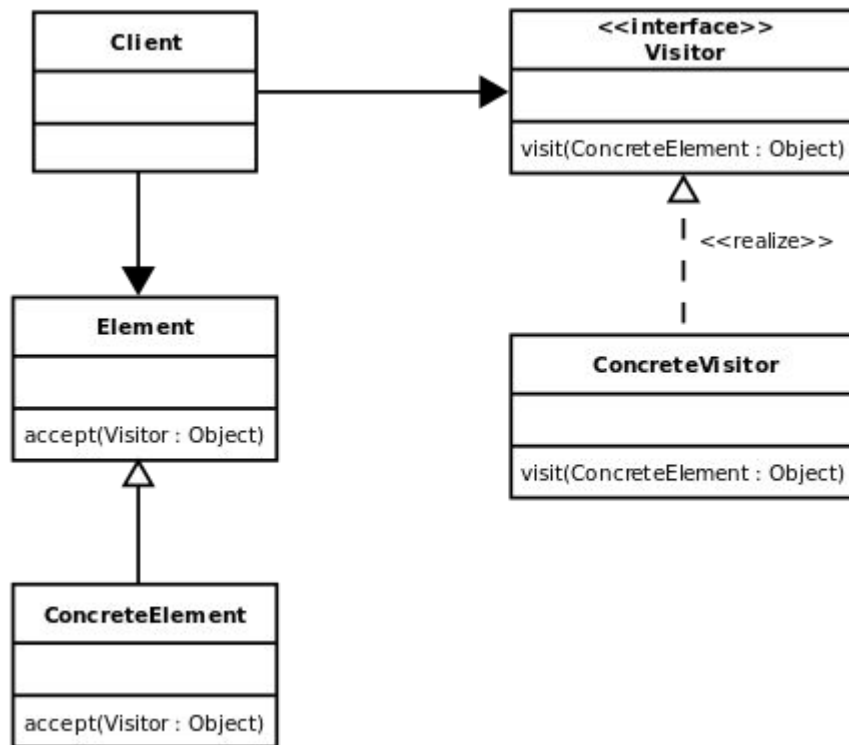
INTERPRETATION / DEPENDENCY INJECTION

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

Más allá de las HOFs

- ❖ Sección 1: type classes y la modularidad
- ❖ Sección 2: Implicits y context bounds
- ❖ Ejercicio 1: Comparable
- ❖ Sección 3: Type classes vs. herencia/adaptadores
- ❖ Sección 4: Type classes vs. factorías abstractas
- ❖ **Sección 5: Type classes vs. visitors**
- ❖ Sección 6: Extensibilidad de datos y funcionalidad
- ❖ Sección 7: Type constructor classes
- ❖ Homework

Patrón *visitor*



Datos + funcionalidad

Representaciones OO y ADTs

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

ADTs

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

```
trait Exp { def eval: Value }
case class Lit(x: Int) extends Exp {
  def eval: Value = VInt(x)
}
case class Add(l: Exp, r: Exp) extends Exp {
  def eval: Value = VInt(l.eval.getInt + r.eval.getInt)
}
```

OO

Type classes vs. ADTs

Funcionalidad

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

ADTs

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

TCs

Evaluación de expresiones

Directa vs. indirecta

directa

```
val expr: Exp =  
    Add(Lit(1), Lit(3))  
  
val v: Value = expr.eval
```

OO

directa

```
val expr: Exp =  
    Add(Lit(1), Lit(3))  
  
val v: Value = eval(expr)
```

ADT

indirecta

```
def expr[E: Exp]: E =  
    lit(3) + lit(7)  
  
val v: Value =  
    adt.eval(expr(ADTExp))
```

TC

directa

```
def expr[E: Exp]: E =  
    lit(3) + lit(7)  
  
val v: Value = expr(eval)
```

TC

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)
```

```
val v: Int = op(eval)
```

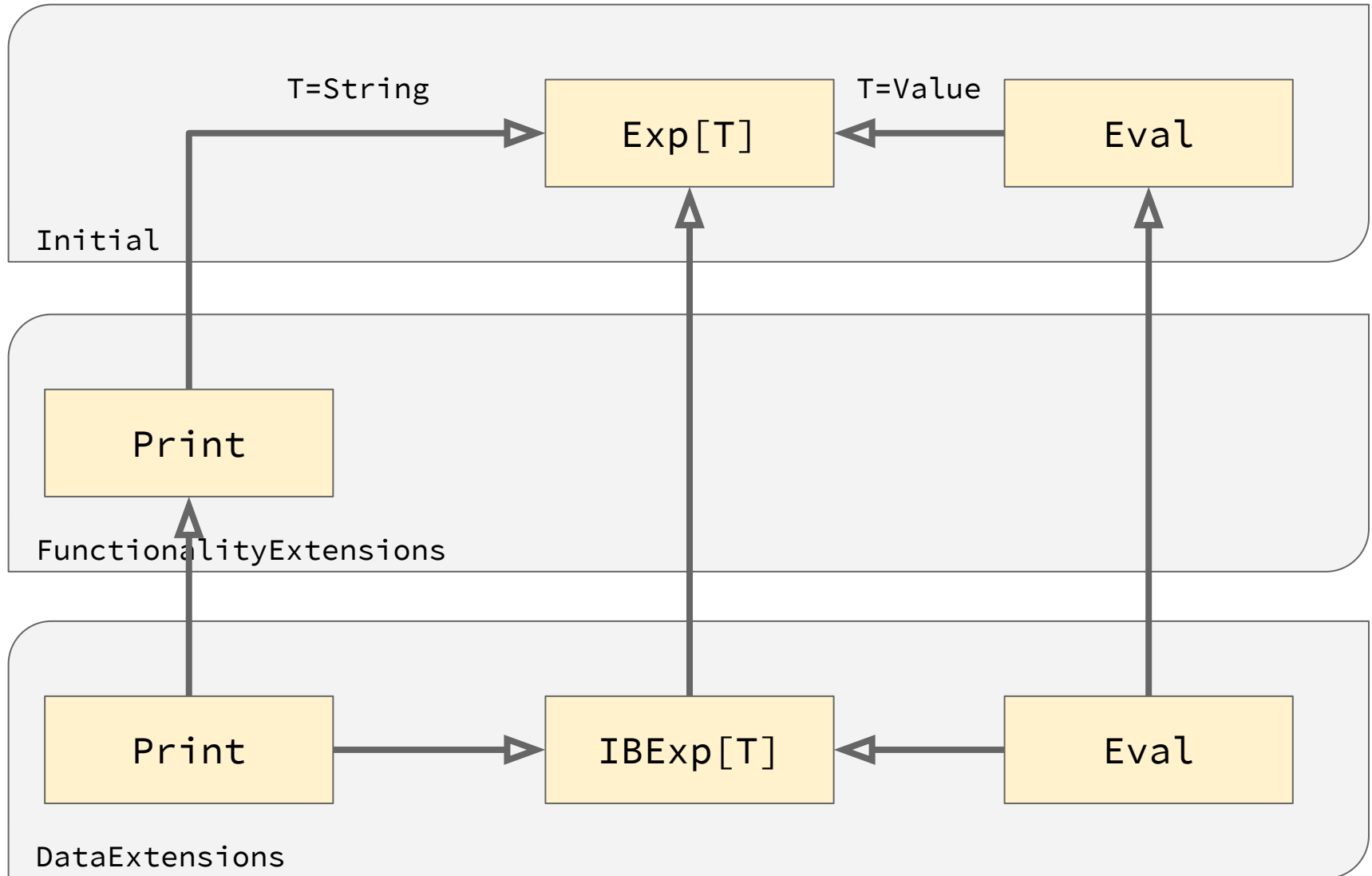

Más allá de las HOFs

- ❖ Sección 1: type classes y la modularidad
- ❖ Sección 2: Implicits y context bounds
- ❖ Ejercicio 1: Comparable
- ❖ Sección 3: Type classes vs. herencia/adaptadores
- ❖ Sección 4: Type classes vs. factorías abstractas
- ❖ Sección 5: Type classes vs. visitors
- ❖ **Sección 6: Extensibilidad de datos y funcionalidad**
- ❖ Sección 7: Type constructor classes
- ❖ Homework

Extensión de datos vs. extensión de funcionalidad

	DATA EXTENSIONS	FUNCTIONALITY EXTENSIONS
Object classes		
ADTs		
Type classes		

Type class extension



Más allá de las HOFs

- ❖ Sección 1: type classes y la modularidad
- ❖ Sección 2: Implicits y context bounds
- ❖ Ejercicio 1: Comparable
- ❖ Sección 3: Type classes vs. herencia/adaptadores
- ❖ Sección 4: Type classes vs. factorías abstractas
- ❖ Sección 5: Type classes vs. visitors
- ❖ Sección 6: Extensibilidad de datos y funcionalidad
- ❖ **Sección 7: 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[?, ?]
```

Unsafety

```
object WithADTs{  
  import ADTExtensions.DataExtensions._  
  
  val estoSiCompila: Exp = Add(Lit(3), Bool(true))  
}
```

ADTs

```
object WithTypeClasses {  
  import TypeclassExtensions.DataExtensions._  
  
  def estoSiCompila[E](implicit E: IntBoolExpr[E]): E =  
    E.add(E.lit(3), E.bool(true))  
}
```

OO

DENIED

ADT safety



```
sealed trait Exp[A]
case class Lit(x: Int) extends Exp[Int]
case class Add(l: Exp[Int], r: Exp[Int]) extends Exp[Int]
case class Bool(b: Boolean) extends Exp[Boolean]
```

```
// Ya no puedo crear expresiones erroneas
// val estoNoCompila: Exp[Int] = Add(Lit(3), Bool(true))
```

ADT safety

```
// ¡Ya no necesitamos el tipo `Value` para el
// intérprete `eval`!
def eval[A](e: Exp[A]): A = e match {
  case Lit(i) => i
  case Add(l, r) => eval(l) + eval(r)
  case Bool(b) => b
}

def print[A](e: Exp[A]): String = e match {
  case Lit(i) => i.toString
  case Add(l, r) => s"${print(l)} + ${print(r)}"
  case Bool(b) => b.toString
}
```


Type class safety



```
trait Expr[E[_]] {  
  def lit(i: Int): E[Int]  
  def add(e1: E[Int], e2: E[Int]): E[Int]  
  def bool(b: Boolean): E[Boolean]  
}  
  
// def estoNoCompila[E[_]](implicit E: Expr[E]): E[Int] =  
//   E.add(E.lit(3), E.bool(true))
```

Type class safety

```
type Id[A] = A
implicit object Eval extends Expr[Id] {
  def lit(i: Int): Int = i
  def add(e1: Int, e2: Int): Int = e1 + e2
  def bool(b: Boolean): Boolean = b
}

type StringF[A] = String
implicit object Print extends Expr[StringF] {
  def lit(i: Int) = i.toString
  def add(e1: String, e2: String) = s"$e1 + $e2"
  def bool(b: Boolean) = b.toString
}
```

Más allá de las HOFs

- ❖ Sección 1: type classes y la modularidad
- ❖ Sección 2: Implicits y context bounds
- ❖ Ejercicio 1: Comparable
- ❖ Sección 3: Type classes vs. herencia/adaptadores
- ❖ Sección 4: Type classes vs. factorías abstractas
- ❖ Sección 5: Type classes vs. visitors
- ❖ Sección 6: Extensibilidad de datos y funcionalidad
- ❖ Sección 7: Type constructor classes
- ❖ **Homework**

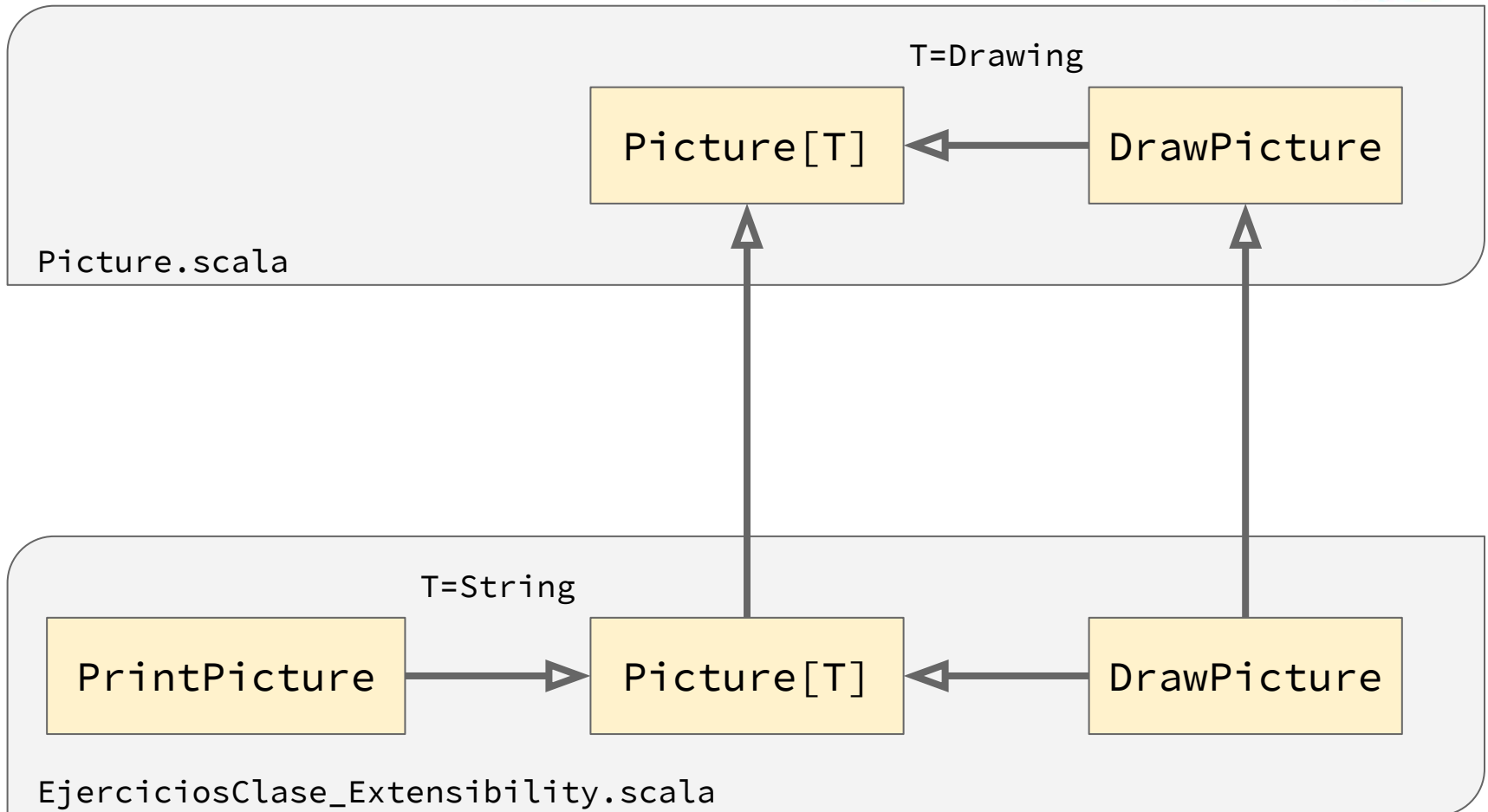


Ejercicios para casa

- **Ej.1** `tema2-.../homework/EjercicioTypeClasses.scala`
 - Impl. *type class* para cálculos estadísticos
 - Comparar con una solución típica de OO (patrón *Adapter*)
- **Ej.2** `tema2-.../homework/EjercicioExtensibility.scala`
 - Extensión de funcionalidad y datos para el ejemplo de *diagrams*.
- **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*

EJERCICIO

Extensión de Picture



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