



# Programación funcional en Scala

## 3. *Funciones puras y lenguajes*



*Habla Computing*  
[info@hablapps.com](mailto:info@hablapps.com)  
[@hablapps](https://twitter.com/hablapps)

# Objetivos

- ¿Qué es una **arquitectura funcional**?
  - ¿En qué se diferencia (y se parece) a una **arquitectura convencional**?
  - ¿Qué **problemas** trata de resolver?
- 
- ¿Qué son las **funciones puras**?
  - ¿Qué papel juegan en todo esto las **type classes**?
  - ¿Y las **mónadas**?
  - ¿Y los **lenguajes específicos de dominio**?

# Funciones puras y lenguajes

- ❖ Sección 1: APIs convencionales: Store
- ❖ Sección 2: APIs funcionales
  - Ejercicio 1: ¿Funciones puras o impuras?
- ❖ Sección 3: APIs funcionales: Store
- ❖ Sección 4: APIs funcionales: IO
  - Ejercicio 2: Programas IO
- ❖ Sección 5: Mónadas y APIs
- ❖ Sección 6: Combinación de efectos
  - Homework 1/2: Programas File IO + IO

# ScalaMAD: Scala Programming @ Madrid

[Home](#)[Members](#)[Sponsors](#)[Photos](#)[Discussions](#)[More](#)[Join us!](#)**Madrid, Spain**

Founded Jun 20, 2013

Scaleros 665

Group reviews 4

Upcoming 1  
Meetups

Past Meetups 19

Our calendar

**Organizers:**

Juan Manuel



Serrano,

Ignacio

Navarro,

Javier Fuentes Sánchez,

Nando Sola

[Contact](#)**We're about:**Programming  
Languages · Computer  
programming · Scala  
Programming · Scala ·  
Functional Programming

Scala es un lenguaje de programación orientado a objetos y, a la vez, es un lenguaje funcional.

[Join us!](#)

- Si el grupo no está moderado, se añade el nuevo miembro al grupo y se notifica a los organizadores
- Si el grupo está moderado, se registra la solicitud, y se notifica a los organizadores
- Si el usuario o el grupo no existe, el usuario ya es miembro, o ya existe una solicitud, la solicitud no tiene ningún efecto

**iBienvenido!**[Upcoming 1](#)[Suggested 0](#)[Past](#)[Calendar](#)**Programación funcional con Scala: al compilador y al abogado, háblales claro**Madrid International Lab  
C/ Bailén, 41 28005 , Madrid ([map](#))Quizás no seas consciente de ello, pero si en tu día a día te encuentras con problemas a la hora de probar, reutilizar, corregir, optimizar o cambiar un programa, muy... [LEARN MORE](#)

Mon Oct 26

7:00 PM

[RSVP](#)

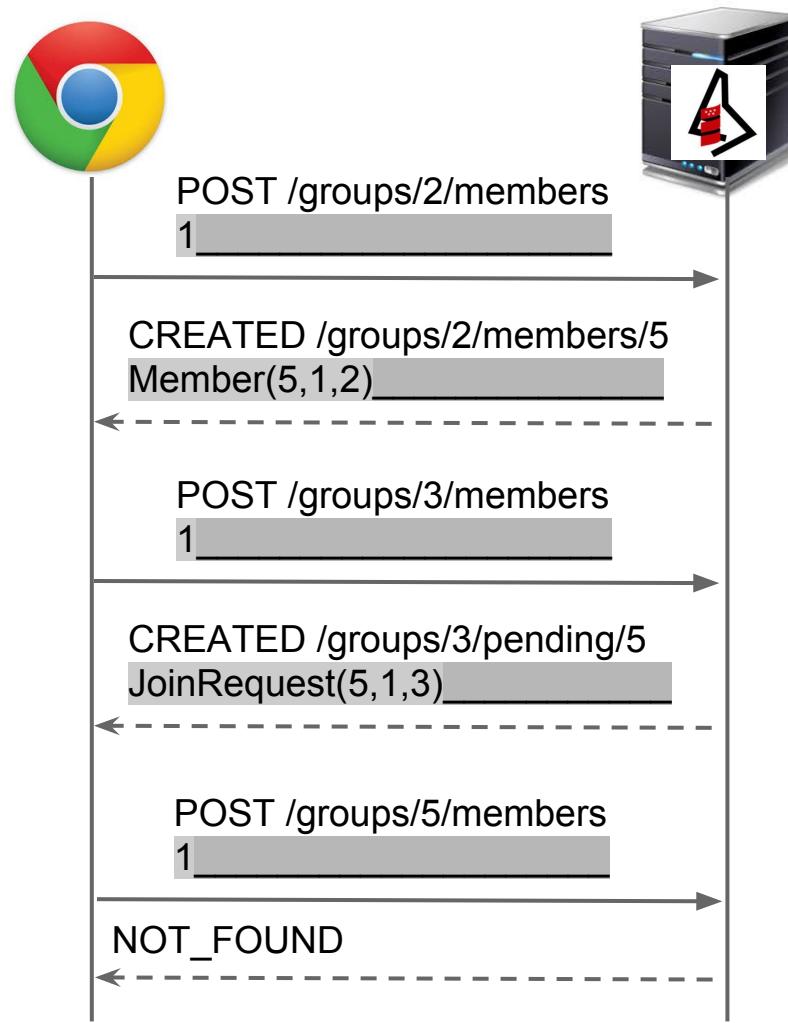
7 days left

91 going

2 comments

**What's new**

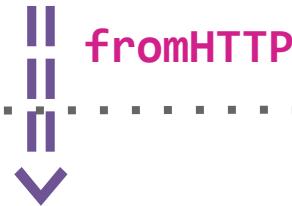
# Diseño RESTful



# Separación de aspectos

## *Presentación vs. lógica de negocio*

PRESENTATION



LOGIC



join

# Join debería centrarse en la lógica de negocio exclusivamente



```
def join(req: JoinRequest): JoinResponse = {  
    “check that the user exists  
    get group data  
    if the group is moderated  
        register request  
    otherwise  
        create new member”  
}
```

# Aspectos entremezclados:



JoinRequest

join



JoinResponse

```
def join(req: JoinRequest): JoinResponse = {  
    // check if user exists  
    val user = DB.withSession { implicit session =>  
        user_table.byId(Some(req.uid)).first  
    }  
  
    ...  
}  
  
case class JoinRequest(jid: Option[Int], uid: Int, gid: Int)
```



# Separación de aspectos: *lógica de negocio vs. persistencia*

```
trait Store{  
    def getGroup(gid: Int): Group  
    def getUser(uid: Int): User  
    def putJoin(join: JoinRequest): JoinRequest  
    def putMember(member: Member): Member  
    ...  
}
```

OO INTERFACES

```
trait NoSQLStore extends Store{  
trait MySQLStore extends Store{  
    ...  
}
```



# Lógica de negocio “purificada”



```
def join(request: JoinRequest): JoinResponse = {  
    val _      = getUser(request.uid)  
    val group = getGroup(request.gid)  
    if (group.must_approve)  
        Left(putJoin(request))  
    else  
        Right(putMember(Member(...)))  
}  
  
type JoinResponse = Either[JoinRequest, Member]
```

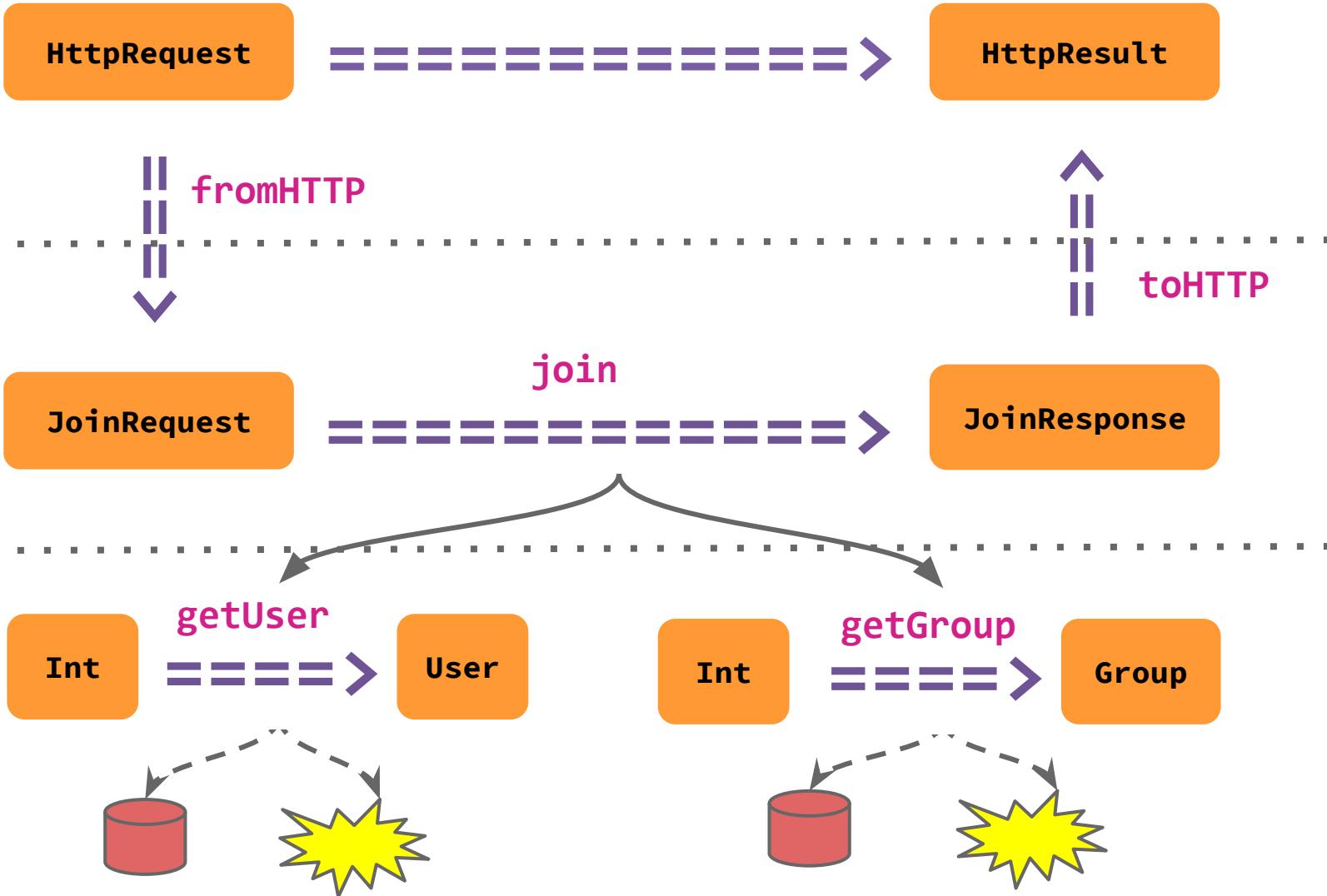
# Aspectos de persistencia relegados al intérprete

```
trait MySQLStore extends Store{  
  
    def getGroup(gid: Int): Group =  
        DB.withSession { implicit session =>  
            group_table.byId(Some(gid)).first  
        }  
  
    def getUser(uid: Int): User =  
        ...  
    def putJoin(join: JoinRequest): JoinRequest =  
        ...  
    def putMember(member: Member): Member =  
        ...  
}
```



# Architectura

PRESENTATION





JoinRequest

==>

JoinResponse

## SIMPLE OO INTERFACES



<<IMPLEMENT.1>>

<<IMPLEMENT.2>>

<<IMPLEMENT.3>>

# ¿Qué tal el desacoplamiento?

## Reto: upgrade de persistencia

```
val result: Group =  
  DB.withSession { implicit session =>  
    group_table.byId(Some(gid)).first  
  }
```



Asynchronous!



```
val result: Future[Group] =  
  db.run(  
    group_table.byId(Some(gid)).result.head  
  )
```

# Refactorización: Intérprete

```
trait MySQLStore extends Store{

    def getGroup(gid: Int): Group = {
        val f: Future[Group] = db.run(group_table....)
        ???
    }
    ...
}
```

# Refactorización: intérprete

```
trait MySQLStore extends Store{  
  
    def getGroup(gid: Int): Group = {  
        val f: Future[Group] = db.run(group_table....)  
        Await.result(f, Duration.Inf)  
    }  
    ...  
}
```



We can't block ...

# Refactorización: intérprete

```
trait MySQLStore extends Store{

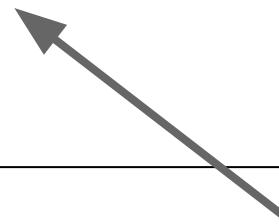
  def getGroup(gid: Int): Future[Group] =
    db.run(group_table.byId(Some(gid)).result.head)

  ...
}
```

So ...

# Refactorización: API

```
trait Store{
    def getGroup(gid: Int): Future[Group]
    def getUser(uid: Int): Future[User]
    def putJoin(join: JoinRequest): Future[JoinRequest]
    def putMember(member: Member): Future[Member]
}
```

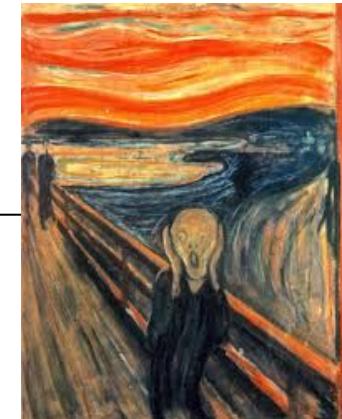


we modify the interface!



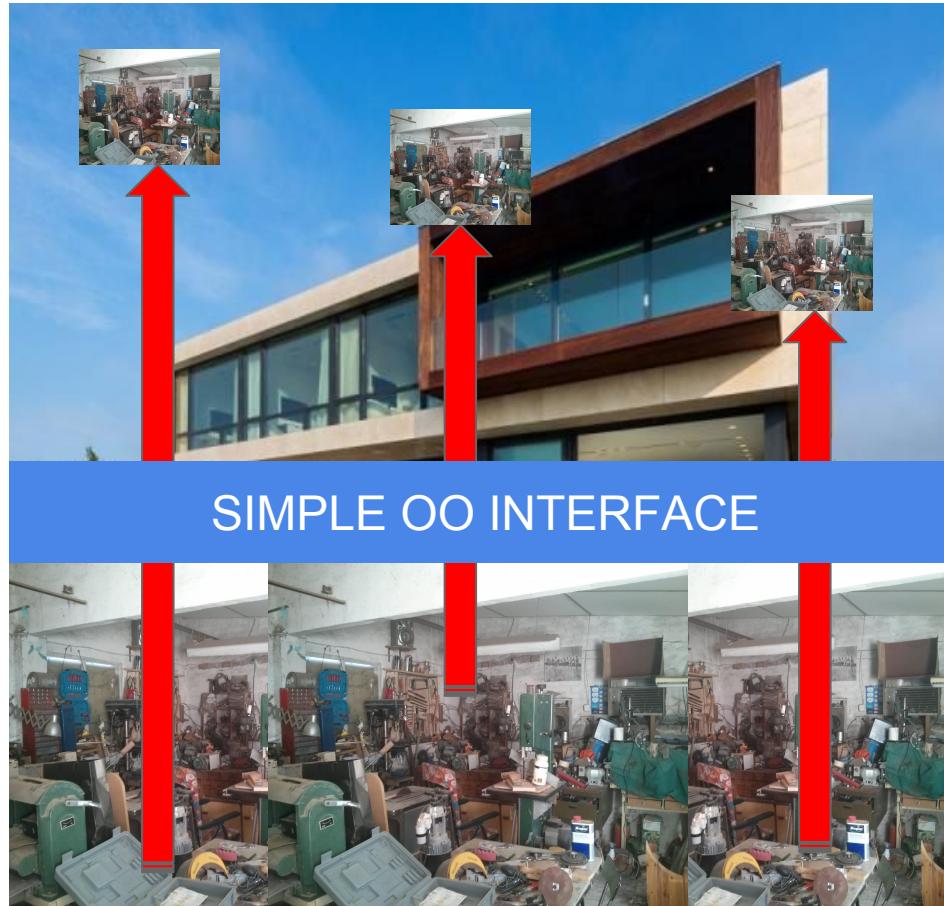
# Refactorización: Lógica de negocio

```
import scala.concurrent.{ExecutionContext, Future}  
import ExecutionContext.Implicits.global  
  
def join(request: JoinRequest): Future[JoinResponse] =  
  ...
```



and the whole business logic!

# Simple OO Interfaces *leak* non-functional concerns



# Otras posibles necesidades no-funcionales

```
trait Store{
    def getGroup(gid: Int): Either[Error, Group]
    def getUser(uid: Int): Either[Error, User]
    ...
}

trait Store{
    def getGroup(gid: Int): State => (State, Group)
    def getUser(uid: Int): State => (State, User)
    ...
}

trait Store{
    def getGroup(gid: Int): Future[Group]
    def getUser(uid: Int): Future[User]
    ...
}
```

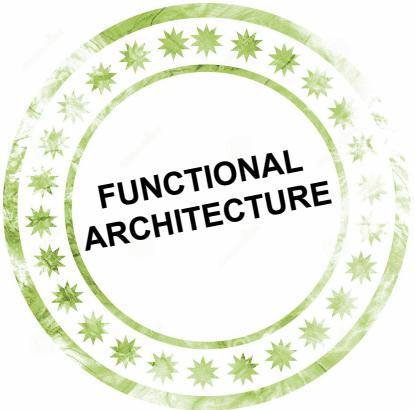
ERROR  
HANDLING

TESTING

ASYNCHRONY

# Funciones puras y lenguajes

- ❖ Sección 1: APIs convencionales: Store
- ❖ **Sección 2: APIs funcionales**
  - Ejercicio 1: ¿Funciones puras o impuras?
- ❖ Sección 3: APIs funcionales: Store
- ❖ Sección 4: APIs funcionales: IO
  - Ejercicio 2: Programas IO
- ❖ Sección 5: Mónadas y APIs
- ❖ Sección 6: Combinación de efectos
  - Ejercicio 3: Programas File IO + IO



APIs  
on steroids

PURE  
FUNCTIONS

SIDE-EFFECT FREE =  
FUNCTIONAL



programs

LANGUAGES

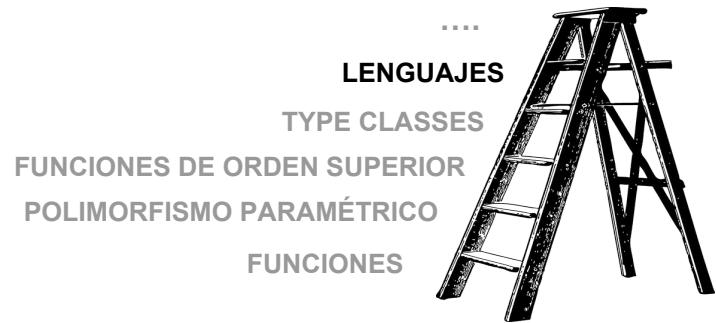
INTERPRETERS

SIDE-EFFECTFUL =  
NON-FUNCTIONAL





# Funciones puras como mecanismo de modularidad



- Tu aplicación se puede descomponer en:
  - **Código libre de efectos colaterales:**
    - Compuesto únicamente de funciones puras
    - Especifican *qué* hacer, i.e. qué efectos se deben ejecutar
  - **Código de efectos colaterales:**
    - Compuesto de funciones impuras
    - A cargo de *ejecutar* los efectos especificados por la parte pura

# Definición (*reminder* Tema 1)

La programación funcional es programar con ***funciones puras***: módulos de software altamente componibles, que reciben valores de entrada, devuelven valores de salida, y no hacen nada más.

# Definición (*formal*)



## Referential transparency and purity

An expression  $e$  is *referentially transparent* if, for all programs  $p$ , all occurrences of  $e$  in  $p$  can be replaced by the result of evaluating  $e$  without affecting the meaning of  $p$ . A function  $f$  is *pure* if the expression  $f(x)$  is referentially transparent for all referentially transparent  $x$ .<sup>3</sup>

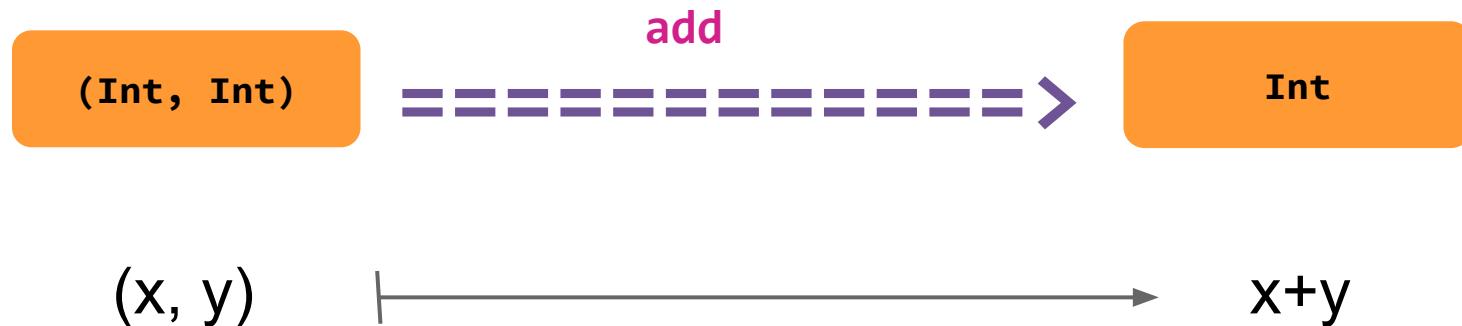
# Si una función es *pura* entonces ...

- El único resultado que podemos *observar* de su ejecución es el valor devuelto por la función: no tiene *efectos colaterales*, solo computa valores
- Permite crear expresiones *referencialmente transparentes*: podemos sustituir la llamada a la función por el valor devuelto y el programa no cambia
- Devuelve siempre el mismo resultado para la misma entrada: son *deterministas* y pueden ser *cacheadas*

# Un ejemplo de función pura

```
def add(x: Int, y: Int): Int =  
    x + y
```

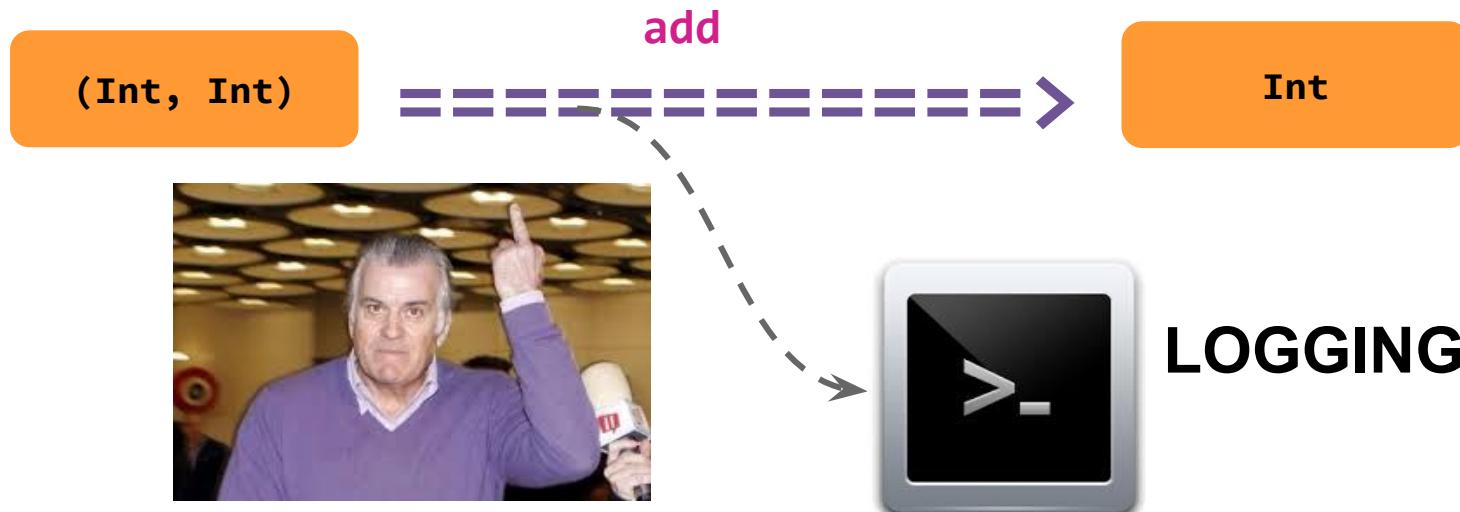
- Ningún efecto observable, solo el resultado



# Un ejemplo de función impura

```
def add(x: Int, y: Int): Int = {  
    val z = x + y  
    println(s"adding $x + $y = $z")  
    z  
}
```

- ¡Sí hay un efecto observable!



# Un ejemplo de función impura

```
def add(x: Int, y: Int): Int = {  
    val z = x + y  
    println(s"adding $x + $y = $z")  
    z  
}
```

- ¿Tiene efecto de lado?
  - ¡Sí! El mensaje de log
- ¿Es *referentially transparent* el resultado?
  - ¡No! Si cambiamos la llamada por el valor que resulta no se producirá el mensaje de log, i.e. cambiará el comportamiento del programa

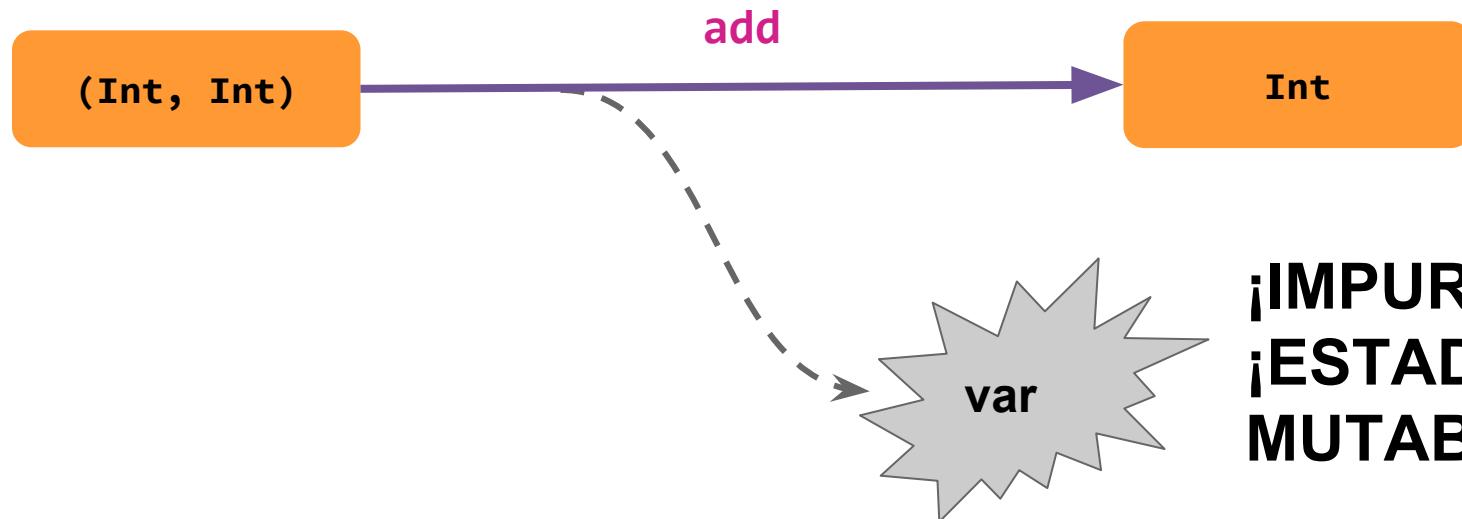
# Funciones puras y lenguajes

- ❖ Sección 1: APIs convencionales: Store
- ❖ Sección 2: APIs funcionales
  - *Ejercicio 1: ¿Funciones puras o impuras?*
- ❖ Sección 3: APIs funcionales: Store
- ❖ Sección 4: APIs funcionales: IO
  - *Ejercicio 2: Programas IO*
- ❖ Sección 5: Mónadas y APIs
- ❖ Sección 6: Combinación de efectos
  - *Ejercicio 3: Programas File IO + IO*

# Ejemplo I: ¿Pura o impura?

```
var log: Seq[String] = Seq.empty

def add(a: Int, b: Int): Int = {
    log = s"Adding $a + $b" +: log
    a+b
}
```



# Ejemplo I: ¿Pura o impura?

```
var log: Seq[String] = Seq.empty

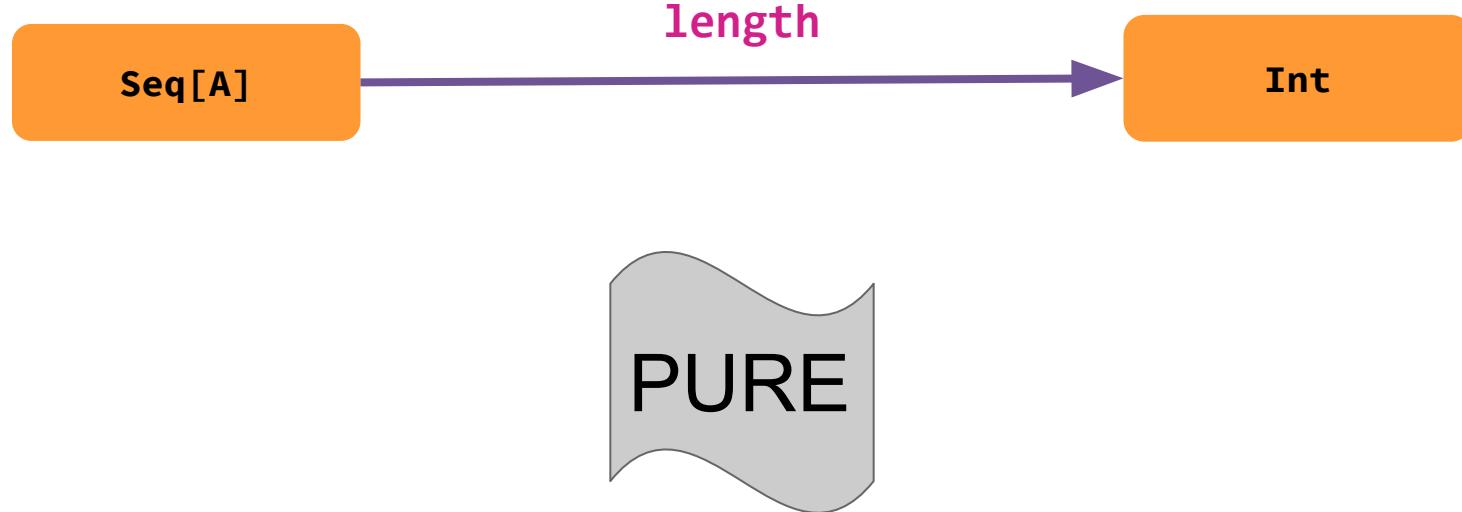
def add(a: Int, b: Int): Int = {
    log = s"Adding $a + $b" +: log
    a+b
}
```

- ¿Tiene efectos de lado?
- ¿Es ref. transparente?
- ¿Siempre devuelve lo mismo?

`var log` es observable desde fuera ya que está fuera del *scope* de la función

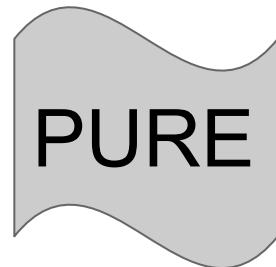
# Ejemplo II: ¿Pura o impura?

```
def length[A](seq: Seq[A]): Int =  
  seq match {  
    case Nil => 0  
    case _ :: xs => length(xs) + 1  
  }
```



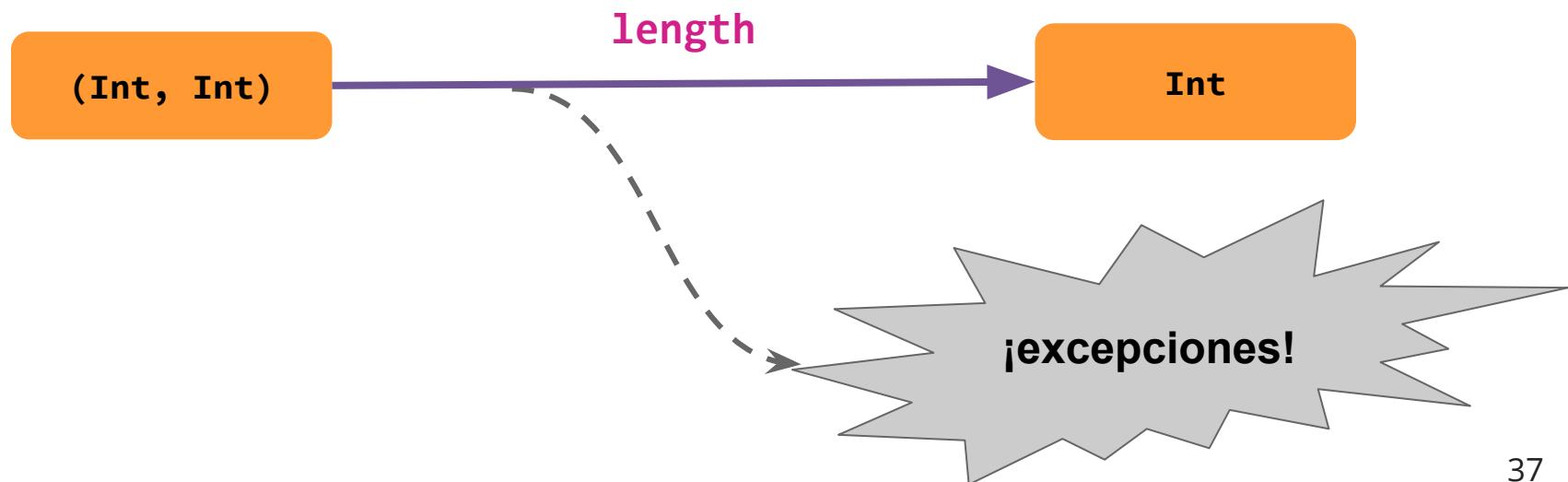
# Ejemplo III: ¿Pura o impura?

```
def length[A](seq: Seq[A]): Int = {  
    var l = 0  
    for (_ <- seq) l += 1  
    l  
}
```



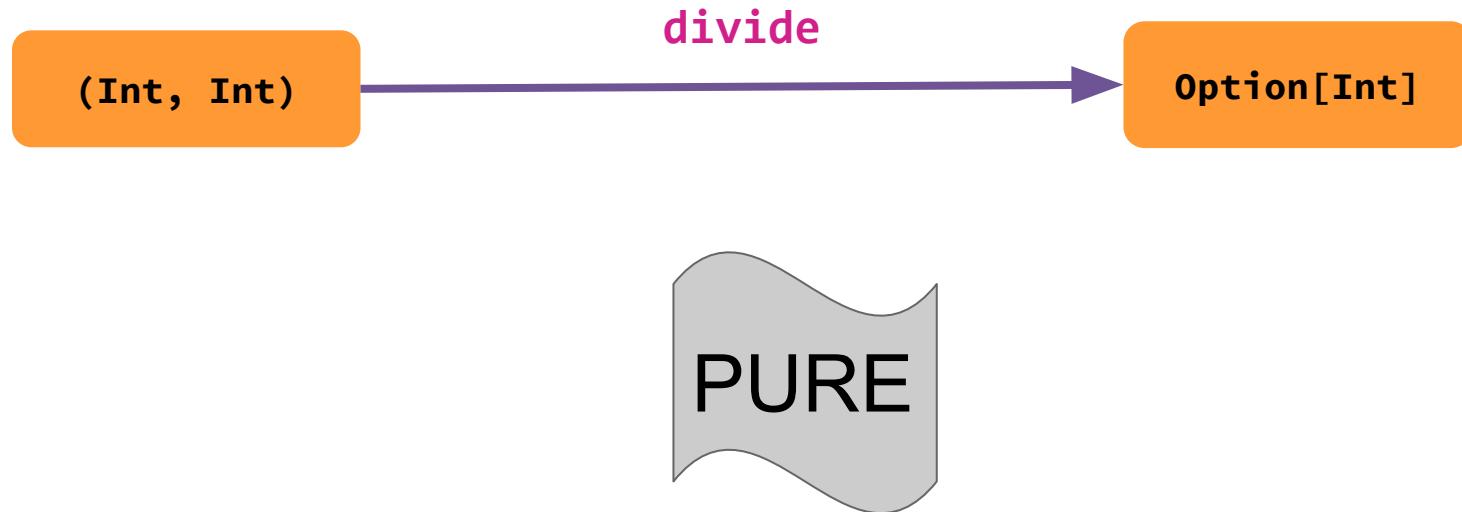
# Ejemplo IV: ¿Pura o impura?

```
def divide(dividend: Int, divisor: Int): Int =  
  if (divisor == 0)  
    throw new RuntimeException(  
      "Sorry, can't divide by 0 :(")  
  else dividend/divisor
```



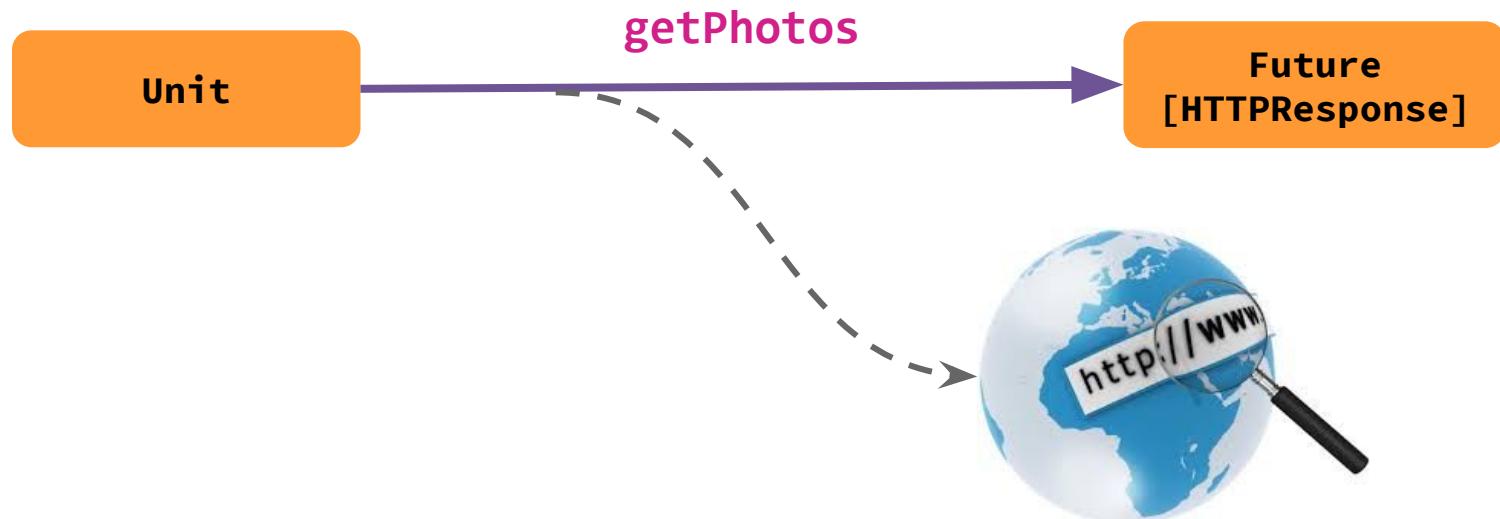
# Ejemplo V: ¿Pura o impura?

```
def divide(dividend: Int, divisor: Int): Option[Int] =  
  if (divisor == 0)  
    None  
  else Option(dividend/divisor)
```



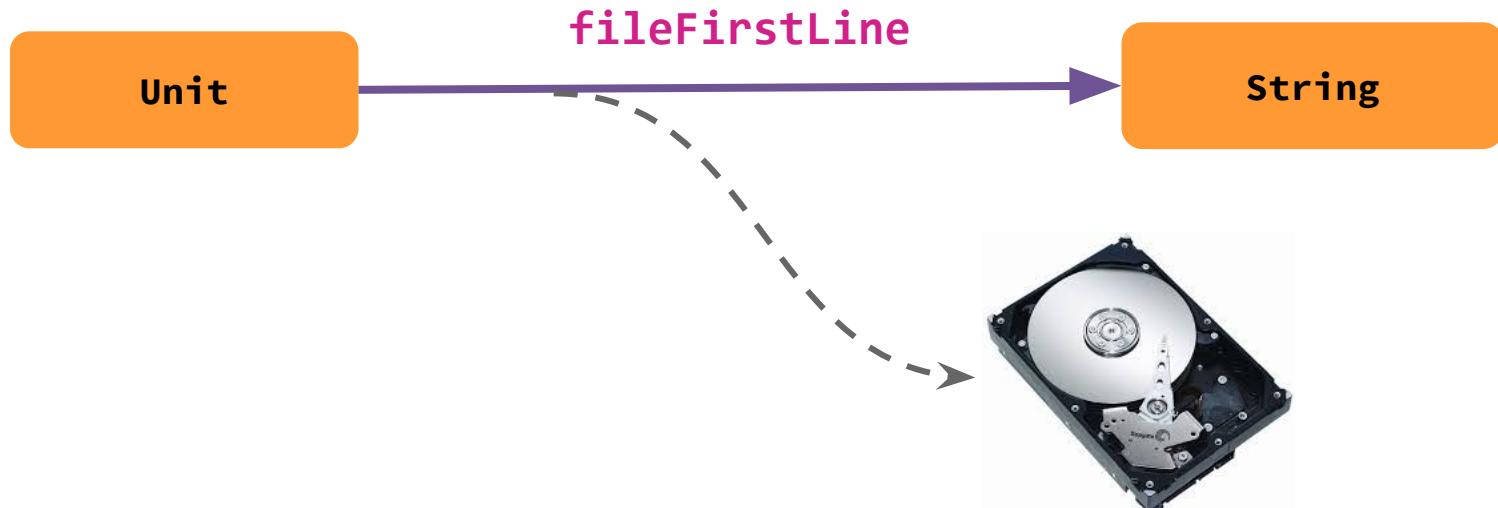
# Ejemplo VI: ¿Pura o impura?

```
def getPhotos: Future[HttpResponse] =  
  send(Get("http://example.com/photos"))("localhost", 80)
```



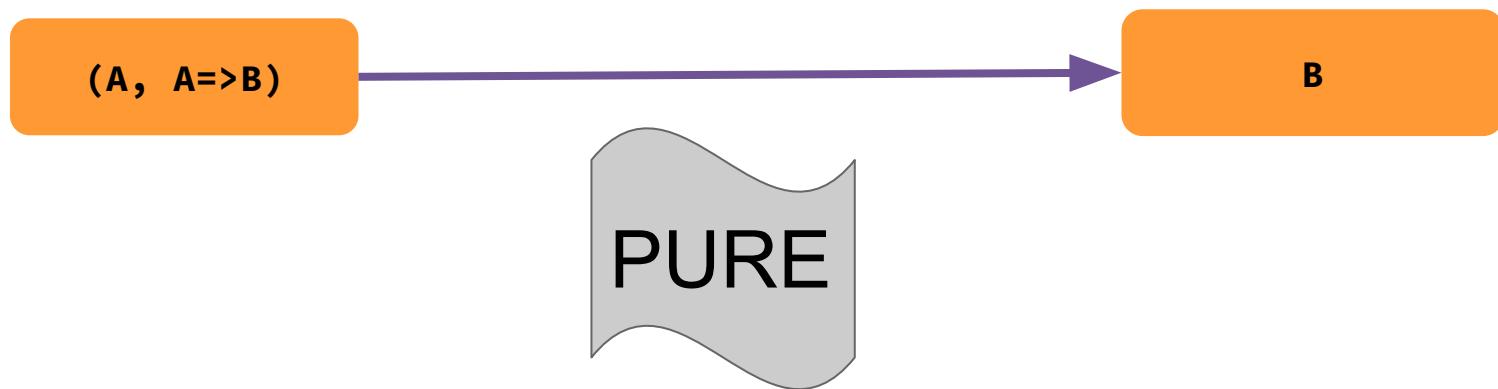
# Ejemplo VII: ¿Pura o impura?

```
def fileFirstLine(path: String): String = {  
    val br = new BufferedReader(new FileReader(path))  
    val line = br.readLine()  
    br.close()  
    line  
}
```



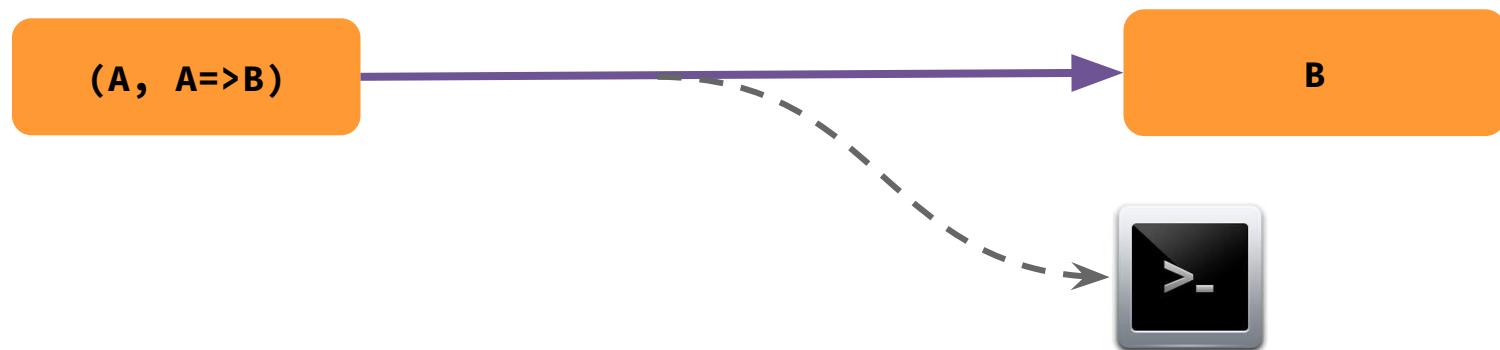
# Ejemplo VIII: ¿Pura o impura?

```
def apply[A, B](a: A)(f: A => B): B =  
  f(a)
```



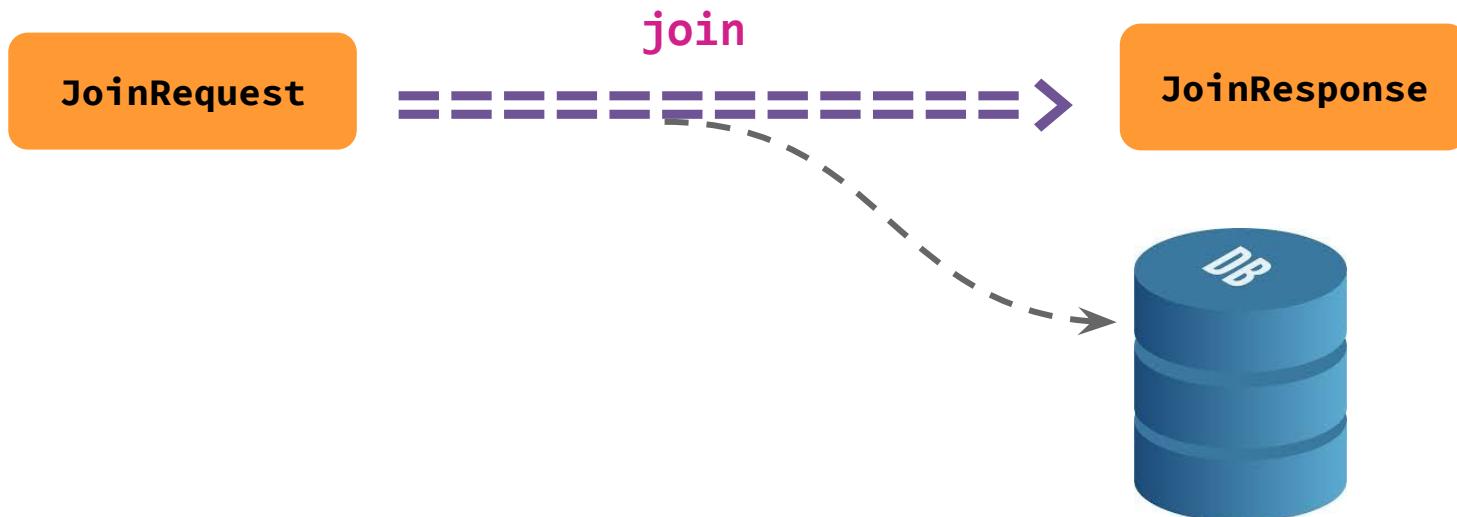
# Ejemplo IX: ¿Pura o impura?

```
def apply[A,B](a: A)(f: A => B): B = {  
    println(s"passing value $a")  
    f(a)  
}
```



# Ejemplo X: ¿Pura o impura?

```
def join(request: JoinRequest): JoinResponse = {  
    val _      = IO.getUser(request.uid)  
    val group = IO.getGroup(request.gid)  
    ...  
}
```



# ¿Qué efectos tienen vuestras aplicaciones?

LOGGING  
ESTADO MUTABLE  
ERRORES

...



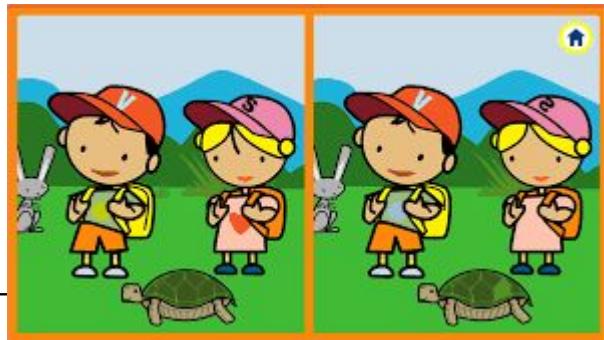
# Funciones puras y lenguajes

- ❖ Sección 1: APIs convencionales: Store
- ❖ Sección 2: APIs funcionales
  - Ejercicio 1: ¿Funciones puras o impuras?
- ❖ **Sección 3: APIs funcionales: Store**
- ❖ Sección 4: APIs funcionales: IO
  - Ejercicio 2: Programas IO
- ❖ Sección 5: Mónadas y APIs
- ❖ Sección 6: Combinación de efectos
  - Ejercicio 3: Programas File IO + IO

# Funciones puras y lenguajes



# ¿Es posible abstraer las diferencias?



```
trait Store{  
    def getGroup(gid: Int): Either[Error, Group]  
    def getUser(uid: Int): Either[Error, User]  
    ...  
}
```

```
trait Store{  
    def getGroup(gid: Int): State => (State, Group)  
    def getUser(uid: Int): State => (State, User)  
    ...  
}
```

```
trait Store{  
    def getGroup(gid: Int): Future[Group]  
    def getUser(uid: Int): Future[User]  
    ...  
}
```

```
trait Store{  
    def getGroup(gid: Int): Group  
    def getUser(uid: Int): User  
    ...  
}
```

# APIs funcionales

*Basadas en type (constructor) classes*

```
trait Store[Program[_]]{
    def getUser(id: Int): Program[User]
    ...
}
```

```
type Failure[T] = Either[Error,T]
object ErrorStore extends Store[Failure]{
    def getUser(id: Int): Either[Error,User] = ...
}
```

```
type State[T] = AppState => (AppState,T)
object StateStore extends Store[State]{...}
```

```
object AsyncStore extends Store[Future]{...}
```

```
type Id[T]=T
object SyncStore extends Store[Id]{...}
```

# APIs funcionales

- La type class define *la clase de los lenguajes* de cierto tipo de efectos
  - Store, IO, File IO, Web services, logging, errores, etc.
  - Las funciones de la API devuelven *programas* escritos en ese lenguaje
- La type class se puede instanciar de forma pura o impura
  - Impura: Id, Future, ...
  - Pura: State, ...

# Solución funcional (lenguaje)

```
trait Store[P[_]]{
    def getGroup(gid: Int): P[Group]
    def getUser(uid: Int): P[User]
    def putJoin(join: JoinRequest): P[JoinRequest]
    def putMember(member: Member): P[Member]
    ...
    // combinators
    def doAndThen[A, B](f: P[A])(cont: A => P[B]): P[B]
    def returns[A](a: A): P[A]
    // derived combinators
    def map[A, B](f: F[A])(m: A=>B): P[B] =
        doAndThen(f)(m andThen returns)
}
```

# Solución funcional (lógica)

```
import Store.Syntax._

def join[P[_]: Store](request: JoinReq): P[JoinRes] = for {
  user   <- getUser(request.uid)
  group  <- getGroup(request.gid)
  result <- Store.cond(
    test = group.must_approve,
    left = putJoin(request),
    right = putMember(Member(...)))
}
} yield result
```

# Solución funcional (intérprete)



```
object Interpreter extends Store[Id]{
    // Operadores de Store
    def putJoin(join: JoinRequest): JoinRequest =
        DB.withSession { implicit session =>
            val maybeId = join_table returning
                join_table.map(_.jid) += join
            join.copy(jid = maybeId)
        }
    ...
    // Operadores de composición

    def returns[A](a: A): A = a
    def doAndThen[A, B](fa: A)(f: A => B): B = f(fa)
}
```

# Solución funcional (intérprete)



```
object FutureStore extends Store[Future] {  
    // Operadores de Store  
    def putJoin(join: JoinRequest): Future[JoinRequest] =  
        db.run((join_table returning join_table.map(_.jid)  
            into ((req,id) => req.copy(jid = id))) += join)  
    ...  
  
    // Operadores de composición  
  
    def returns[A](a: A): Future[A] = Future(a)  
    def doAndThen[A, B](fa: Future[A])(  
        f: A => Future[B]): Future[B] = fa.flatMap(f)  
}
```

## TYPE (CONS.) CLASS SIGNATURE / LANGUAGE INTERFACE / FUNCTIONAL API

```
trait Store[P[_]]{  
    def getGroup(gid: Int): P[Group]  
    ...  
}
```

## (AD-HOC) POLYMORPHIC FUNCTION / FUNCTIONS OVER API

```
def join[P[_]: Store](  
    request: JoinReq): P[JoinRes] = for{  
    _ <- getUser(request.uid)  
    ...  
} yield result
```

## TYPE CLASS INSTANCE / INTERPRETER INTERFACE IMPLEMENTATION

```
object FutureStore  
    extends Store[Future] {  
    def getGroup(gid: Int):  
        Future[Group] = ...  
    ...  
}
```

```
object IdStore  
    extends Store[Id] {  
    def getGroup(gid: Int):  
        Group = ...  
    ...  
}
```

## INTERPRETATION / DEPENDENCY INJECTION

```
val f: Future[JoinResponse] =  
    join(request)(FutureStore)
```

```
val f: JoinResponse =  
    join(request)(IdStore)
```

# Funciones puras y lenguajes

- ❖ Sección 1: APIs convencionales: Store
- ❖ Sección 2: APIs funcionales
  - Ejercicio 1: ¿Funciones puras o impuras?
- ❖ Sección 3: APIs funcionales: Store
- ❖ **Sección 4: APIs funcionales: IO**
  - Ejercicio 2: Programas IO
- ❖ Sección 5: Mónadas y APIs
- ❖ Sección 6: Combinación de efectos
  - Ejercicio 3: Programas File IO + IO

# APIs funcionales

## *Input/Output*

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



```
trait IO[P[_]]{  
    def read: P[String]  
    def write(msg: String): P[Unit]  
    ...  
}
```

# Funciones puras y lenguajes

- ❖ Sección 1: APIs convencionales: Store
- ❖ Sección 2: APIs funcionales
  - Ejercicio 1: ¿Funciones puras o impuras?
- ❖ Sección 3: APIs funcionales: Store
- ❖ Sección 4: APIs funcionales: IO
  - Ejercicio 2: Programas IO
- ❖ Sección 5: Mónadas y APIs
- ❖ Sección 6: Combinación de efectos
  - Ejercicio 3: Programas File IO + IO

# Ejercicio 2

Implementar programas puros de entrada/salida en base al interfaz IO. [tema3-lenguajes/exercise2](#)

```
def writeANumber: Unit = {  
    val num = IO.read  
    IO.write(evenOdd(num))  
}
```



```
def writeANumber[F[_]: IO]: F[Unit] =  
    ???
```



# Funciones puras y lenguajes

- ❖ Sección 1: APIs convencionales: Store
- ❖ Sección 2: APIs funcionales
  - Ejercicio 1: ¿Funciones puras o impuras?
- ❖ Sección 3: APIs funcionales: Store
- ❖ Sección 4: APIs funcionales: IO
  - Ejercicio 2: Programas IO
- ❖ **Sección 5: Mónadas y APIs**
- ❖ Sección 6: Combinación de efectos
  - Ejercicio 3: Programas File IO + IO

# La type class Mónada

```
trait Monad[M[_]]{
  def flatMap[A, B](fa: M[A])(f: A => M[B]): M[B]
} def returns[A](value: A): M[A]
```

# Las mónadas dan soporte a un estilo de programación *imperativo*

teniendo en cuenta el resultado  
de la acción anterior

```
flatMap[A,B]: F[A] => (A => F[B]) => F[B]
```

Haz algo

y luego, algo  
más

# ¿Qué es un lenguaje monádico?

- La type class *Monad* define la clase de los lenguajes imperativos
  - *flatMap* nos permite crear programas secuenciales
  - *returns* nos permite terminar la secuenciación

```
def program[P[_]: Monad](a: String): P[Boolean] = for {  
    i <- returns(a.length)  
    j <- returns(i + 1)  
} yield j > 0
```

# API funcional Monádica

```
trait Store[P[_]]{
    def getGroup(gid: Int): P[Group]
    def getUser(uid: Int): P[User]
    def putJoin(join: JoinRequest): P[JoinRequest]
    def putMember(member: Member): P[Member]
    ...
}
```

```
trait Monad[M[_]]{
    def flatMap[A, B](fa: M[A])(f: A => M[B]): M[B]
    def returns[A](value: A): M[A]
```

# Servicio funcional Monádico

```
import Store.Syntax._, Monad.Syntax._

def join[P[_]: Store: Monad](request: JoinReq): P[JoinRes] =
  for {
    user   <- getUser(request.uid)
    group <- getGroup(request.gid)
    result <- Store.cond(
      test = group.must_approve,
      left = putJoin(request),
      right = putMember(Member(...)))
  } yield result
```

# Intérprete funcional

## Monádico

```
object IdStore extends Store[Id]{
    // Operadores de Store
    def getGroup(gid: Int): Group =
    def getUser(uid: Int): User =
    def putJoin(join: JoinRequest): JoinRequest =
    ...
}
```

```
object IdMonad extends Monad[Id]{
    def returns[A](a: A): A = a
    def doAndThen[A, B](fa: A)(f: A => B): B = f(fa)
}
```

# Funciones puras y lenguajes

- ❖ Sección 1: APIs convencionales: Store
- ❖ Sección 2: APIs funcionales
  - Ejercicio 1: ¿Funciones puras o impuras?
- ❖ Sección 3: APIs funcionales: Store
- ❖ Sección 4: APIs funcionales: IO
  - Ejercicio 2: Programas IO
- ❖ Sección 5: Mónadas y APIs
- ❖ **Sección 6: Combinación de efectos**
  - Ejercicio 3: Programas File IO + IO

# APIs funcionales

## *File System*

```
trait FS{  
    def readFile(path: String): String  
    def writeFile(path: String)(contents: String): Unit  
    def deleteFile(path: String): Unit  
}
```



```
trait FS[F[_]]{  
    def readFile(path: String): F[String]  
    def writeFile(path: String)(contents: String): F[Unit]  
    def deleteFile(path: String): F[Unit]  
}
```

# APIs funcionales

## Combinación de efectos

```
def copy(orig: String, dest: String): Unit = {  
    write(s"copying $orig into $dest")  
    val content: String = readfile(orig)  
    writefile(dest)(content)  
}
```



```
def copy[F[_]: FS: IO: Monad](  
    orig: String, dest: String): F[Unit] = for {  
    _ <- write(s"copying $orig into $dest")  
    content <- readfile(orig)  
    _ <- writefile(dest)(content)  
} yield ()
```

# Funciones puras y lenguajes

- ❖ Sección 1: APIs convencionales: Store
- ❖ Sección 2: APIs funcionales
  - Ejercicio 1: ¿Funciones puras o impuras?
- ❖ Sección 3: APIs funcionales: Store
- ❖ Sección 4: APIs funcionales: IO
  - Ejercicio 2: Programas IO
- ❖ Sección 5: Mónadas y APIs
- ❖ **Sección 6: Combinación de efectos**
  - Homework 1/2: Programas File IO + IO

# Homework 1

Implementar programas puros que hagan uso de la interfaz del sistema de ficheros (FS)  
**tema3-lenguajes/homework1/Homework1.scala**



# Homework 2

Implementar una terminal de comandos **tema3-lenguajes/homework2/Homework2.scala**

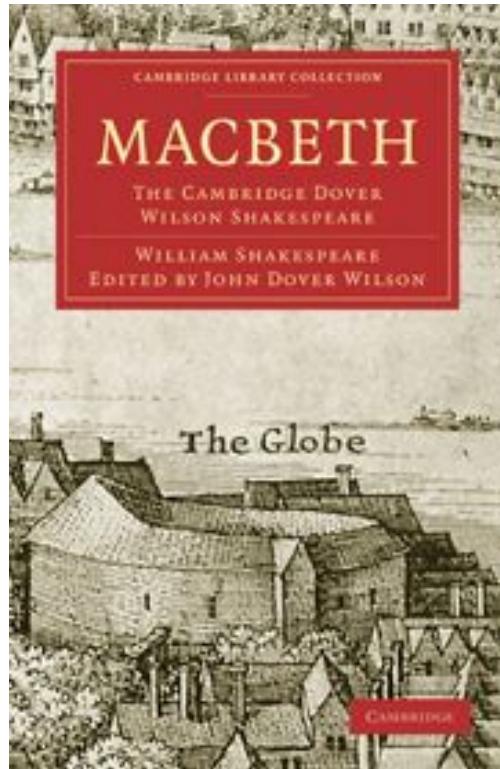
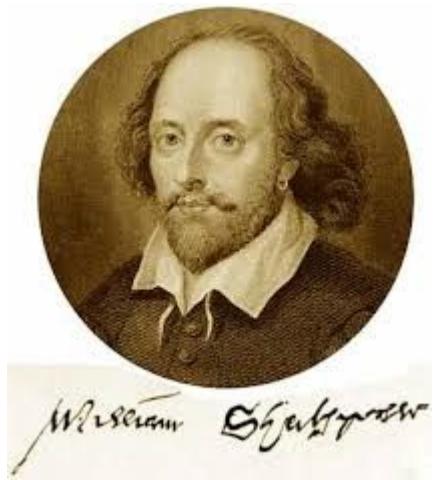
```
trait Commands[Program[_]] {
    type Command = Program[Unit]

    implicit def IO: IO[Program]
    implicit def FS: FS[Program]
    implicit def M: Monad[Program]

    def help: Command = ???
    def cp(args: Seq[String]): Command = ???
    def mv(args: Seq[String]): Command = ???
    def rm(args: Seq[String]): Command = ???
    ...
}
```

# Conclusión

- ¿Se puede utilizar la programación funcional en *cualquier* dominio?
  - Sí, en cualquier dominio es necesario determinar qué es lo que hay que hacer
- ¿Puede ser programada funcionalmente *cualquier* cosa?
  - No, los efectos colaterales tienen que programarse de otra forma (e.g. actores)



# Conclusión

- La programación funcional está estrechamente relacionada con los **lenguajes específicos de dominio** (DSLs)
  - DSLs para tus propios efectos: “your business your language”
- y con la **metaprogramación**
  - Las funciones puras devuelven *programas* que deben ser interpretados o compilados
  - Estos lenguajes intermedios deben ser puros, pero en última instancia será necesario la



Karlos Arguiñano prepara unas codornices guisadas envueltas en tocineta con jamón picado. Para acompañar, tomates cubiertos con provenzal asados en el horno.

#### Ingredientes (4 personas):

- 8 codornices
- 4 tomates
- 8 lonchas de tocineta
- 2 cebollas
- 1 loncha de jamón serrano gruesa
- 250 ml de vino blanco
- 2 dientes de ajo
- 100 gr de pan rallado
- aceite de oliva virgen extra
- sal
- pimienta
- tomillo
- perejil

Elaboración de la receta de Codornices guisadas con tomates a la provenzal:



# ¿Qué otras analogías puedes encontrar?

Pure function	Effectful program	Effect language	Impure code
Composer	Musical score	Musical notation	Interpreter
Play writer	Play	Natural language	Actor
Chef	Recipe	Natural language, drawings, ...	Cook
...	...	...	...



# Conclusión

- ¿Son los paradigmas **imperativo** y **funcional** incompatibles?
  - Simon Peyton Jones (co-creador de Haskell):  
“Haskell is the world’s finest imperative programming language.”
  - La diferencia está simplemente en la sintaxis utilizada:
    - Programas imperativos impuros: sintaxis abstracta de Scala
    - Programas imperativos puros: sintaxis propia

# Conclusión

- ¿Por qué programar funcionalmente?
  - Desacoplamiento entre especificación e infraestructura
    - Mejor reutilización
    - Mejor mantenibilidad
  - Mejores pruebas
  - Mejor agilidad
  - Mejoras organizativas
    - Programadores funcionales ligados a estructuras de negocio