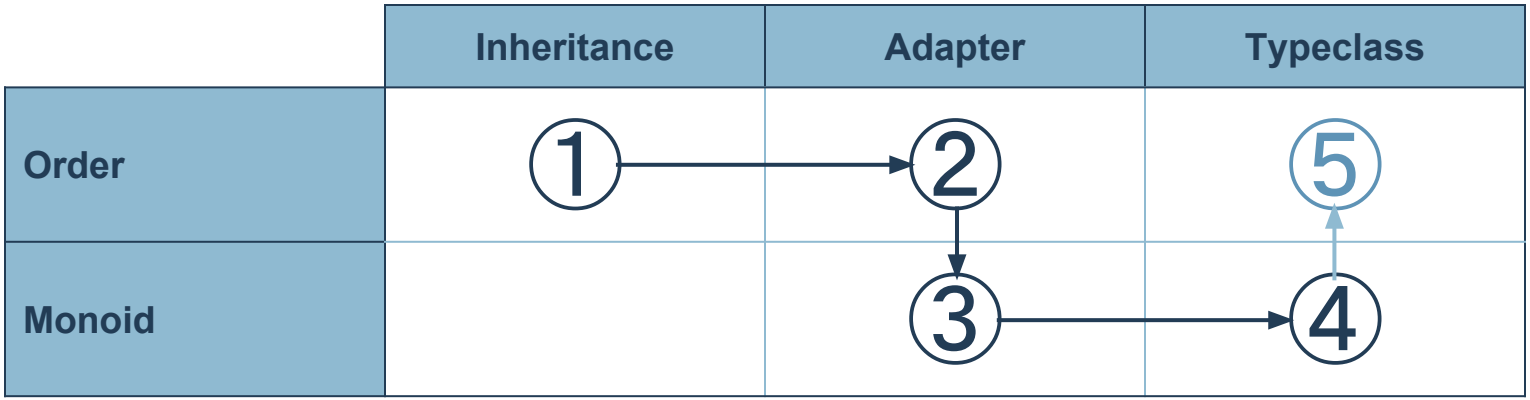
Functional Programming in Scala

Codemotion 2017

Habla Computing
info@hablapps.com
@hablapps

Habla Computing
javier.fuentes@hablapps.com
@javifdev

Outline



Outline

	Inheritance	Adapter	Typeclass
Order	❶	②	⑤
Monoid		③	④

Problem 1: Order

```
def greatest[A](l: List[A]): Option[A]

scala> greatest(List(
     |   Person("Ana", 28),
     |   Person("Berto", 35),
     |   Person("Carlos", 18)))
res0: Option[Person] = Some(Person(Berto,35))

scala> greatest(List(28, 35, 1))
res1: Option[Int] = Some(35)

scala> greatest(List("Berto", "Carlos", "Ana"))
res2: Option[String] = Some("Carlos")
```


You know nothing, Jon Snow.

Approach 1: Inheritance

```scala
trait Order[A] {
  def compare(other: A): Int

  def >(other: A): Boolean = compare(other) > 0
  def ===(other: A): Boolean = compare(other) == 0
  def <(other: A): Boolean = compare(other) < 0
}

def greatest[A <: Order[A]](l: List[A]): Option[A] =
  l.foldLeft(Option.empty[A]) {
    case (Some(max), a) if a < max => Option(max)
    case (_, a) => Option(a)
  }
```

Approach 1: Inheritance

```
case class Person(name: String, age: Int) extends Order[Person] {
  def compare(other: Person) = age - other.age
}

scala> greatest(List(
     |   Person("Ana", 28),
     |   Person("Berto", 35),
     |   Person("Carlos", 18)))
res0: Option[Person] = Some(Person(Berto,35))
```

INTERFACE

```
trait Order[A] {
  def compare(other: A): Int

  def >(other: A): Boolean = compare(other) > 0
  def ===(other: A): Boolean = compare(other) == 0
  def <(other: A): Boolean = compare(other) < 0
}
```

GENERIC FUNCTION

```
def greatest[A <: Order[A]](l: List[A]): Option[A] =
  l.foldLeft(Option.empty[A]) {
    case (Some(max), a) if a < max => Option(max)
    case (_, a) => Option(a)
  }
```

INTERFACE IMPLEMENTATION

```
case class Person(name: String, age: Int)
    extends Order[Person] {
  def compare(other: Person) = age - other.age
}
```

EXECUTION

```
greatest(List(
  Person("Ana", 28),
  Person("Berto", 35),
  Person("Carlos", 18)))
// res0: Option[Person] = Some(Person(Berto,35))
```

Approach 1: Inheritance

```
scala> greatest(List(2, 3, 1))
<console>:14: error: inferred type arguments [Int] do not conform to
method greatest's type parameter bounds [A <: Order[A]]
       greatest(List(2, 3, 1))
       ^
<console>:14: error: type mismatch;
 found   : List[Int]
 required: List[A]
       greatest(List(2, 3, 1))
```

2nd Issue

- What do we do for types out of our control?
  - Primitive types like `Int`, `String`, `Boolean`, ...
  - Third party library types: `DateTime`, ...
- Adapters to the rescue!

Outline

	Inheritance	Adapter	Typeclass
Order	① →	②	⑤
Monoid		③	④

Approach 2: Adapter

```
def greatest[A](l: List[A])(wrap: A => Order[A]): Option[A] =
  l.foldLeft(Option.empty[A]) {
    case (Some(max), a) if wrap(a) < max => Option(max)
    case (_, a) => Option(a)
  }

scala> case class IntOrder(unwrap: Int) extends Order[Int] {
     |   def compare(other: Int) = unwrap - other
     | }
defined class IntOrder

scala> greatest(List(2, 3, 1))(IntOrder(_))
res2: Option[Int] = Some(3)
```

INTERFACE

```
trait Order[A] {
  def compare(other: A): Int

  def >(other: A): Boolean = compare(other) > 0
  def ===(other: A): Boolean = compare(other) == 0
  def <(other: A): Boolean = compare(other) < 0
}
```

GENERIC FUNCTION

```
def greatest[A](l: List[A])(
    wrap: A => Order[A]): Option[A] =
  l.foldLeft(Option.empty[A]) {
    case (Some(max), a) if wrap(a) < max =>
      Option(max)
    case (_, a) => Option(a)
  }
```

INTERFACE IMPLEMENTATION

```
case class IntOrder(unwrap: Int)
    extends Order[Int] {
  def compare(other: Int) = unwrap - other
}
```

EXECUTION

```
greatest(List(2, 3, 1))(IntOrder(_))
// res2: Option[Int] = Some(3)
```


abc

Outline

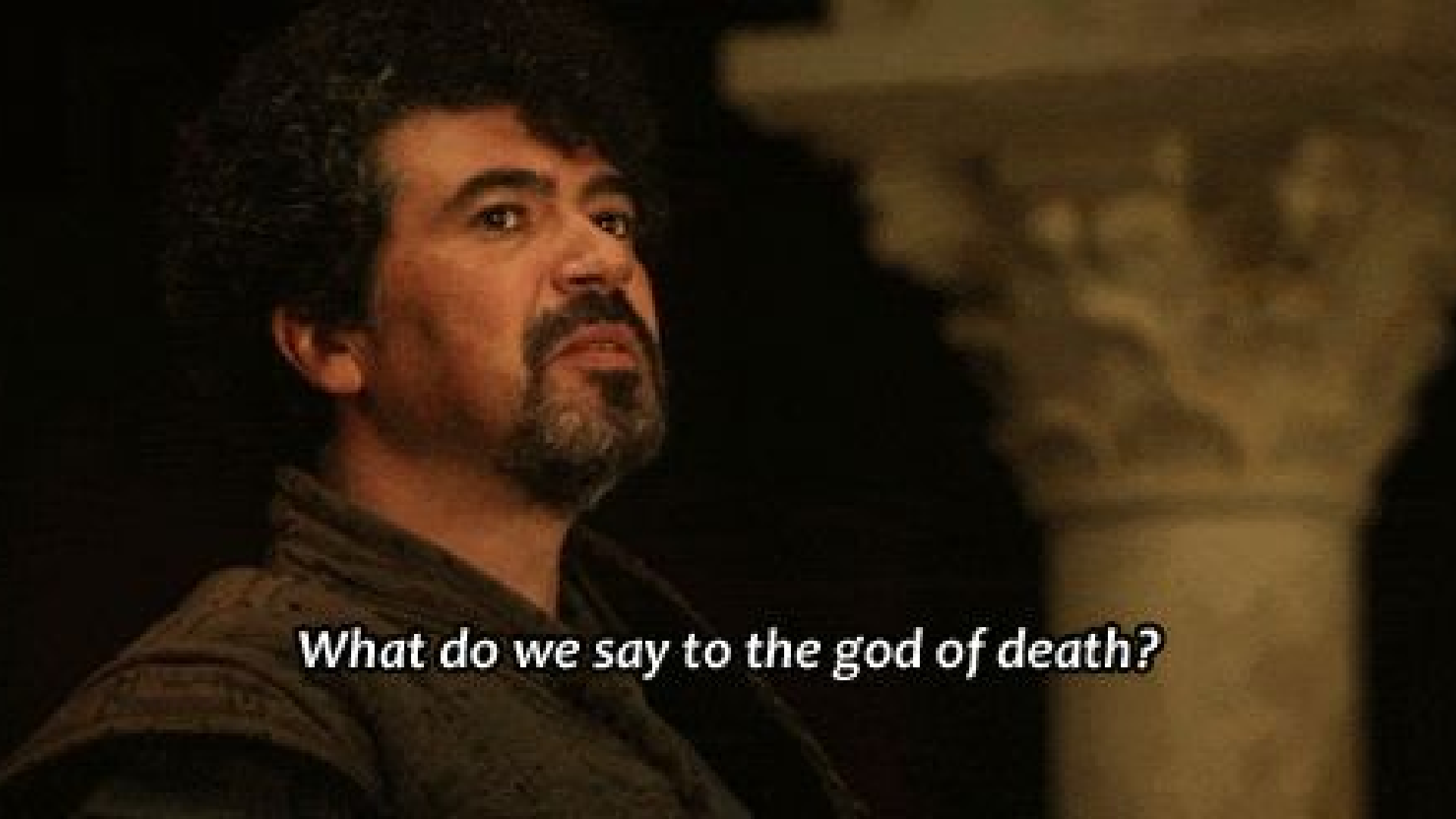
	Inheritance	Adapter	Typeclass
Order	①	②	⑤
Monoid		③	④

Problem 2: Monoid

```
def collapse[A](l: List[A]): A

scala> collapse(List(1, 2, 3, 4))
res0: Int = 10

scala> collapse(List("hello", ", ", "world!"))
res1: String = hello, world!
```

Approach 1: Adapter

```
trait Monoid[A] {
  def empty: A
  def combine(other: A): A
}

def collapse[A](l: List[A])(wrap: A => Monoid[A]): A =
  l.foldLeft[A](???)((a1, a2) => wrap(a1).combine(a2))
```


What do we say to the god of death?

Outline

	Inheritance	Adapter	Typeclass
Order	①	②	⑤
Monoid		③ →	④

Approach 2: Typeclass

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

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

Approach 2: Typeclass

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

scala> collapse(List(1, 2, 3, 4))(intSumMonoid)
res3: Int = 10
```

Approach 2: Typeclass

```
val intMulMonoid: Monoid[Int] =
  new Monoid[Int] {
    val empty: Int = 1
    def combine(i1: Int, i2: Int): Int = i1 * i2
  }

scala> collapse(List(1, 2, 3, 4))(intMulMonoid)
res4: Int = 24
```

Approach 2: Typeclass

```
val stringMonoid: Monoid[String] =
  new Monoid[String] {
    val empty: String = ""
    def combine(s1: String, s2: String): String = s1 + s2
  }

scala> collapse(List("hello", ", ", "world!"))(stringMonoid)
res5: String = hello, world!
```

INTERFACE

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

GENERIC FUNCTION

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

INTERFACE IMPLEMENTATION

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

EXECUTION

```
collapse(List(1, 2, 3, 4))(intSumMonoid)
// res3: Int = 10
```


I do know some things.

Decoupling analysis

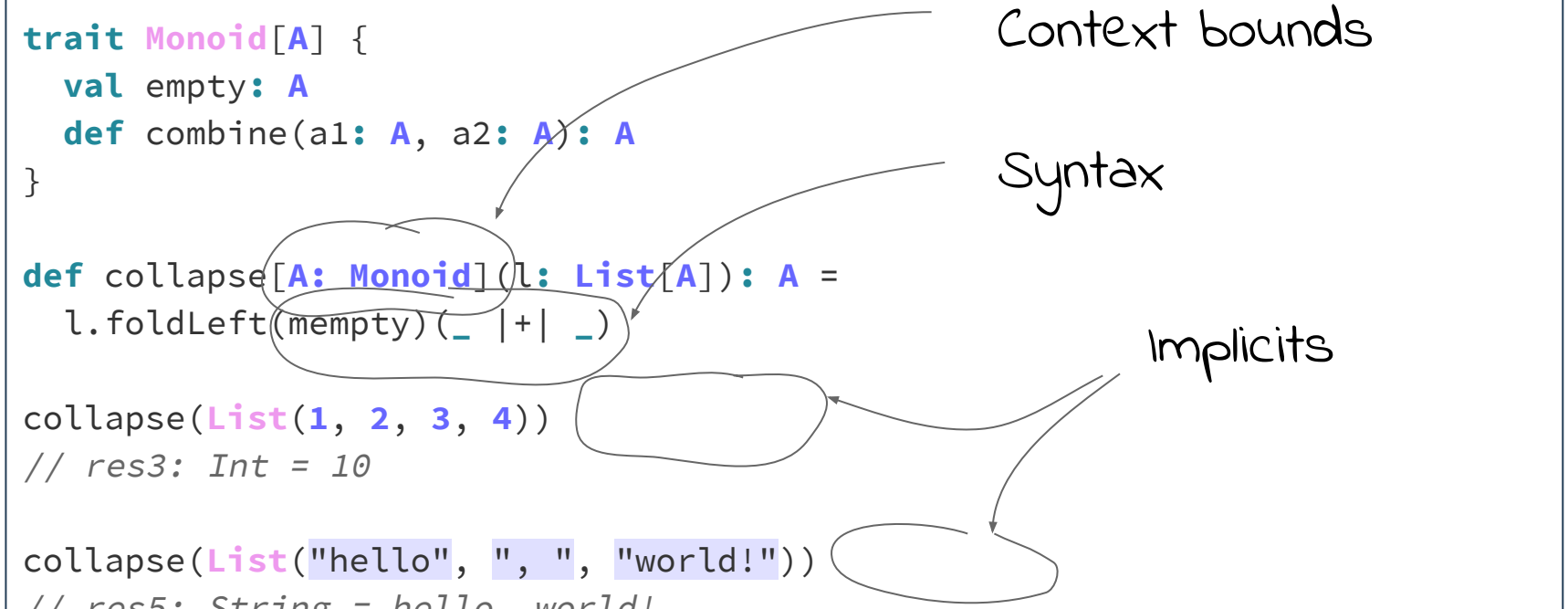
	Inheritance	Adapter	Typeclass
Types	✗	✔	✔
Instances	✗	✗	✔

Conclusions

- Typeclasses are more efficient & modular
- Inheritance can be easier and more straightforward, but at the cost of being much less modular
- Adapters are a try to fix some of these inheritance issues, but also fails at decoupling behavior from values

Typeclass true power (final form)

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

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

collapse(List(1, 2, 3, 4))
// res3: Int = 10

collapse(List("hello", ", ", "world!"))
// res5: String = hello, world!
```



Outline

	Inheritance	Adapter	Typeclass
Order	①	②	⑤
Monoid		③	④


take
the bull
by the
HABLA