Ionuț G. Stan · BJUG #14

Agenda

- What’s Scala
- Why Scala
- Quick language overview

Martin Odersky

- Scala’s designer
- Helped design Java’s generics (Generic Java)
- Worked on the current javac compiler

Martin Odersky

Generic Java team (left to right): Philip Wadler, Martin Odersky, Gilad Bracha, David Stoutamire.

Martin Odersky

Generic Java team (left to right): Philip Wadler, Martin Odersky, Gilad Bracha, David Stoutamire.

What is Scala

- A language for the JVM
- Statically typed
- Compiles to JVM bytecode, just like Java
- Research on compiling to other platforms
- The JVM has influenced Scala's semantics
- Language + compiler + standard library

What is Scala

- Supports two programming paradigms
- Object-Oriented Programming (OOP)
- Functional Programming (FP)
- Created to explore this mix
- Martin Odersky sees FP and OOP as orthogonal
- OOP does not imply imperative

What is FP

- Various opinions
- All centered around the same concept
- FP is programming w/ functions/values
- FP is programming w/o side-effects/mutations
- FP’s roots are in mathematical functions
- A function’s output is determined **entirely** by its input

Why FP

- Easier to go concurrent/parallel
- Easier to reason about code
- Fosters composability

Scala and FP

- Encourages immutability (syntax + stdlib)
- Concise syntax for function literals
- Function types
- Pattern matching
- Case classes
- Favours expressions over statements
- Lazy evaluation

Why Scala

- Improves support for OOP
- Good support for FP
- Runs on JVM, one of the best VMs around
- A better Java
- Yet interoperable with Java
- Powerful type system (vs. Clojure)
- Favours immutability, but allows mutability

Quick Language Overview

Language Features

- Type inference
- Function types and literals
- Objects as functions
- Case classes
- Pattern matching
- Traits
- Lazy evaluation
- Macros (compile-time metaprogramming)

Hello World

```
package ro.bjug.fourteen

object Main {
  def main(args: Array[String]): Unit = {
    val name: String = args(0)
    println(s"Hello $name!")
  }
}
```

Type Inference

```
package ro.bjug.fourteen

object Main {
  def main(args: Array[String]) = {
    val name = args(0)
    println(s"Hello $name!")
  }
}
```

Objects as Functions

```
object Factorial {
  def apply(n: Int): Int = {
    if (n < 2) {
      return 1
    } else {
      return n * Factorial(n - 1)
    }
  }
}

Factorial(5)
```

Expressions

```
object Factorial {
  def apply(n: Int): Int = {
    if (n < 2) 1 else n * Factorial(n - 1)
  }
}

Factorial(5)
```

Function Literals

```
val double: Int => Int = (n)        => n * 2
val double             = (n: Int) => n * 2

double(5)
```

Functions as Objects

```scala
val double = (n: Int) => n * 2
val square = (n: Int) => n * n

double(square(2))
// equivalent to
square.andThen(double)(2)
```

REPL

igstan — ~ — bash — 87×21

~

```
λ ~ master: scala
Welcome to Scala version 2.10.2 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_21).
Type in expressions to have them evaluated.
Type :help for more information.

scala> val repl = "REPL stands for: Read Eval Print Loop"
repl: String = REPL stands for: Read Eval Print Loop

scala> val double = (n: Int) => n * 2
double: Int => Int = <function1>

scala> double(3)
res0: Int = 6

scala> :q
λ ~ master:
```

Standard Library

igstan — ~ — java — 87×21

```
scala> val numbers = List(1, 2, 3, 4, 5, 6)
numbers: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> numbers.filter(n => n % 2 == 0)
res0: List[Int] = List(2, 4, 6)

scala> numbers.filter(_ % 2 == 0).map(_ * 2)
res1: List[Int] = List(4, 8, 12)

scala> numbers.filter(_ % 2 == 0).map(_ * 2).drop(1)
res2: List[Int] = List(8, 12)

scala> numbers.filter(_ % 2 == 0).map(_ * 2).drop(1).take(1)
res3: List[Int] = List(8)

scala> numbers.filter(_ % 2 == 0).map(_ * 2).drop(1).take(1).head
res4: Int = 8

scala>
```

Classes

```scala
class Person(name: String, age: Int) {
  def bio: String =
    s"$name is a $age years old developer."
}

val person = new Person("Ionuț", 28)

person.bio // Ionuț is a 28 years old developer.
```

Companion Object

```scala
class Person(name: String, age: Int) {
  def bio: String =
    s"$name is a $age years old developer."
}

object Person {
  def apply(name: String, age: Int) = {
    new Person(name, age)
  }
}

val person = Person("Ionuț", 28)
```

Companion Object

```scala
class Person(val name: String, val age: Int) {
  def bio: String =
    s"$name is a $age years old developer."
}

object Person {
  def apply(name: String, age: Int) = {
    new Person(name, age)
  }
}

val person = Person("Ionuț", 28)
```

Case Classes

```
case class Person(name: String, age: Int)

val p1 = Person("Ionuț", 28)
val p2 = Person("Ionuț", 28)
```

Case Classes

```
scala> p1.name
res1: String = Ionuț

scala> p2.age
res2: Int = 28

scala> p1 == p2
res3: Boolean = true

scala> p1.hashCode == p2.hashCode
res4: Boolean = true
```

Pattern Matching

```scala
val s = "bar"

s match {
  case "foo" => 1
  case "bar" => 2
  case _     => 3
}
```

Pattern Matching

```
case class Person(name: String, age: Int)

val person = Person("Ionuț", 28)

person match {
  case Person(name, age) =>
    s"$name is $age old."
}
```

Pattern Matching

```
case class City(name: String, country: String)
case class Address(street: String, number: String, city: City)
case class Person(name: String, age: Int, address: Address)

val p = Person(
  name    = "Ionuț",
  age     = 28,
  address = Address(
    number = "42",
    street = "Endless",
    city   = City("Bucharest", "Romania")
  )
)

p match {
  case Person(name, _, Address(_, _, City(city, _)))
    => s"$name lives in $city."
}
```

Pattern Matching

```
case class Person(name: String, age: Int)

val person = Person("Ionuț", 28)

person match {
  case person: Person =>
    s"${person.name} is ${person.age} old."
}
```

A Better Null

```
sealed trait Option[+A]
case object None extends Option[Nothing]
case class Some[A](value: A) extends Option[A]

object Option {
  def apply[A](value: A): Option[A] = {
    if (value == null) None else Some(value)
  }
}
```

A Better Null

```
val user: Option[User] = Users.find(id)

user match {
  case Some(user) => OK(render(user))
  case None       => NotFound()
}

// or using higher-order functions

user.map(u => OK(render(u))).getOrElse {
  NotFound()
}
```

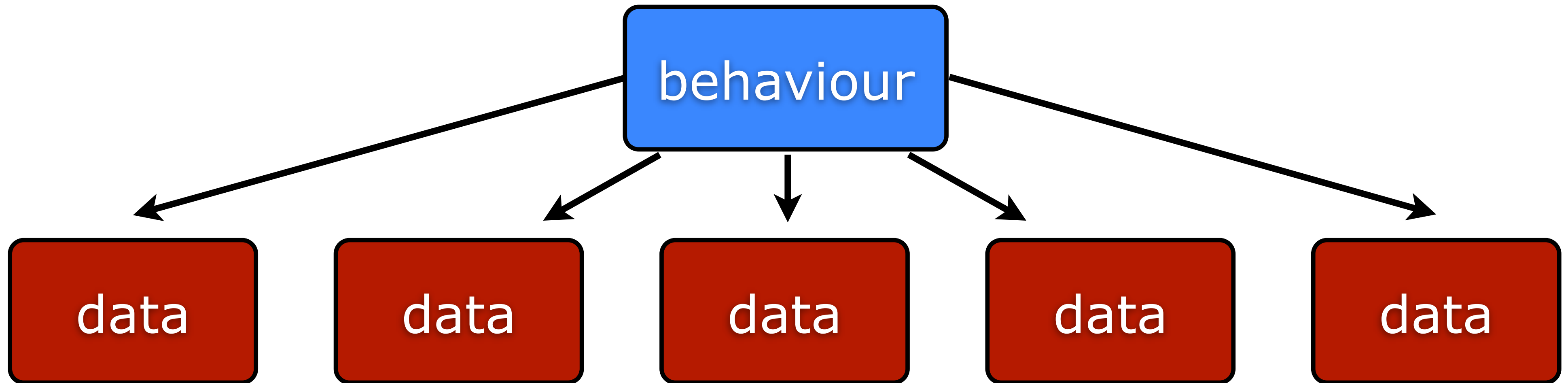
Pattern Matching

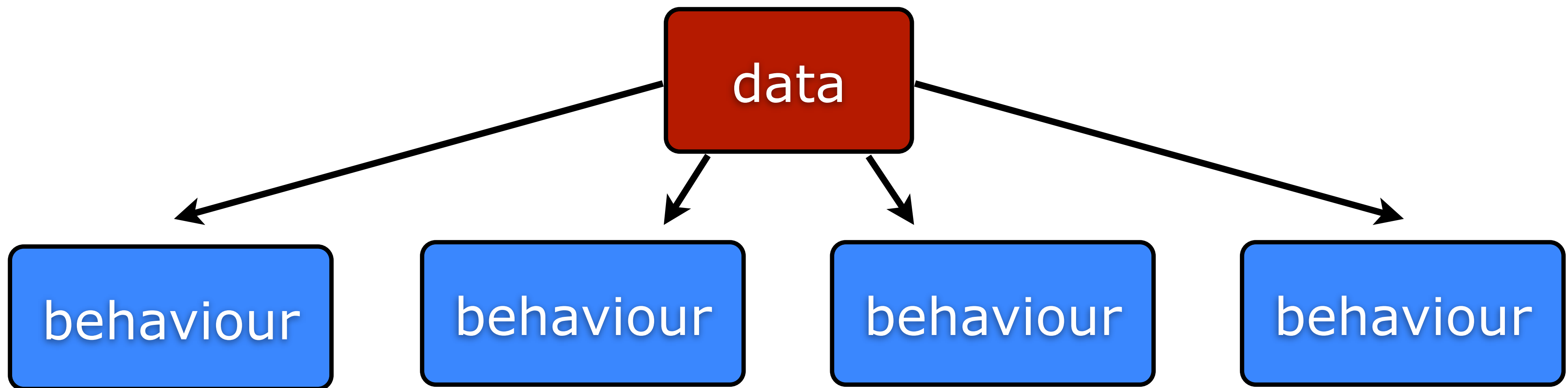
```
"ionut.g.stan@gmail.com" match {
  case Email(user, domain) => println(domain)
  case _                   => println("Invalid email.")
}
```

Pattern Matching

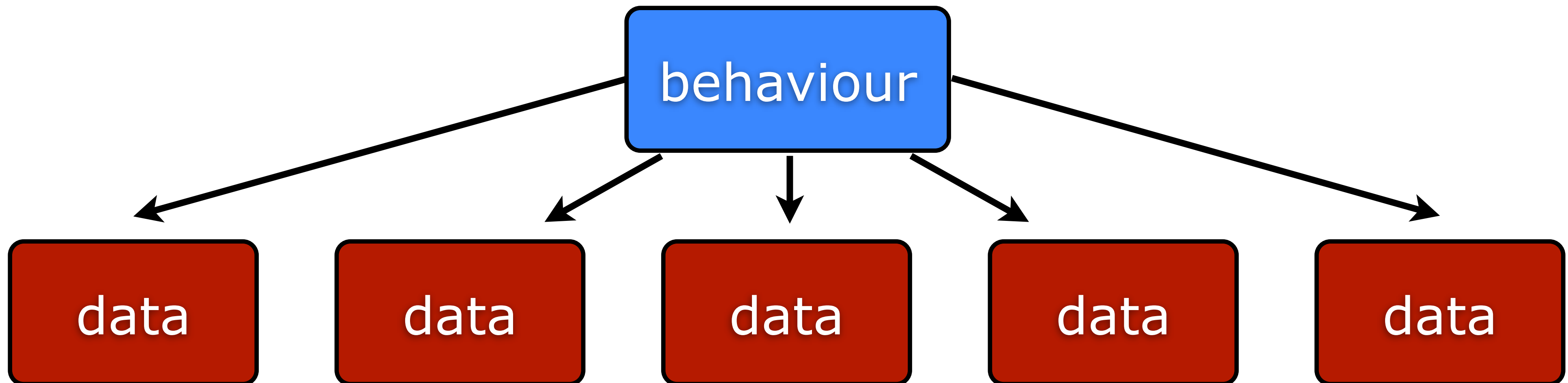
```
object Email {
  def unapply(email: String) = {
    if (valid(email)) {
      // Another form of pattern matching, in assignment.
      val Array(user, domain) = email.split('@')
      Some( (user, domain) )
    } else None
  }

  private def valid(email: String) = {
    email.count(char => char == '@') == 1
  }
}

"ionut.g.stan@gmail.com" match {
  case Email(user, domain) => println(domain)
  case _                   => println("Invalid email.")
}
```

OOP vs. FP





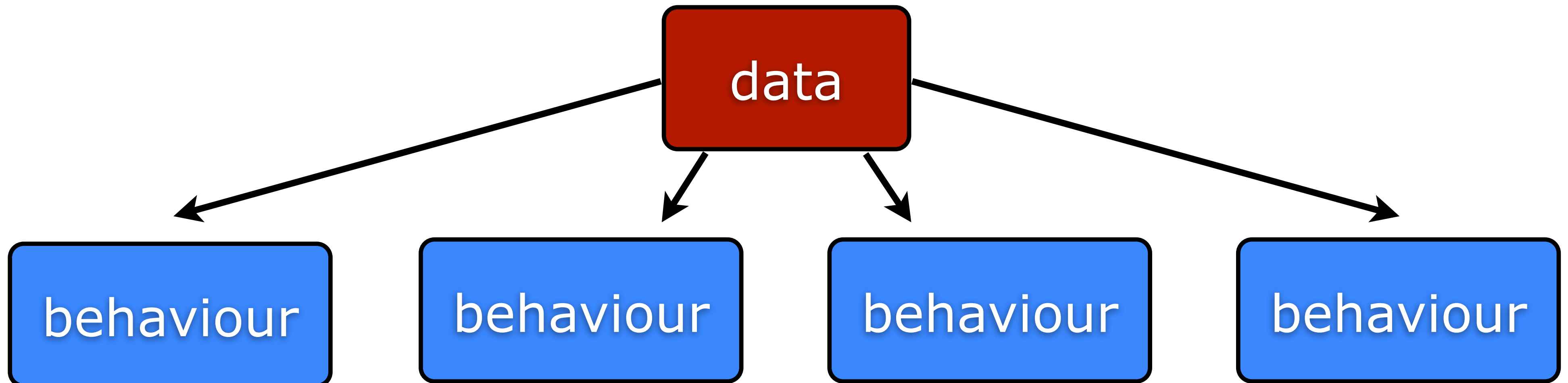
OOP



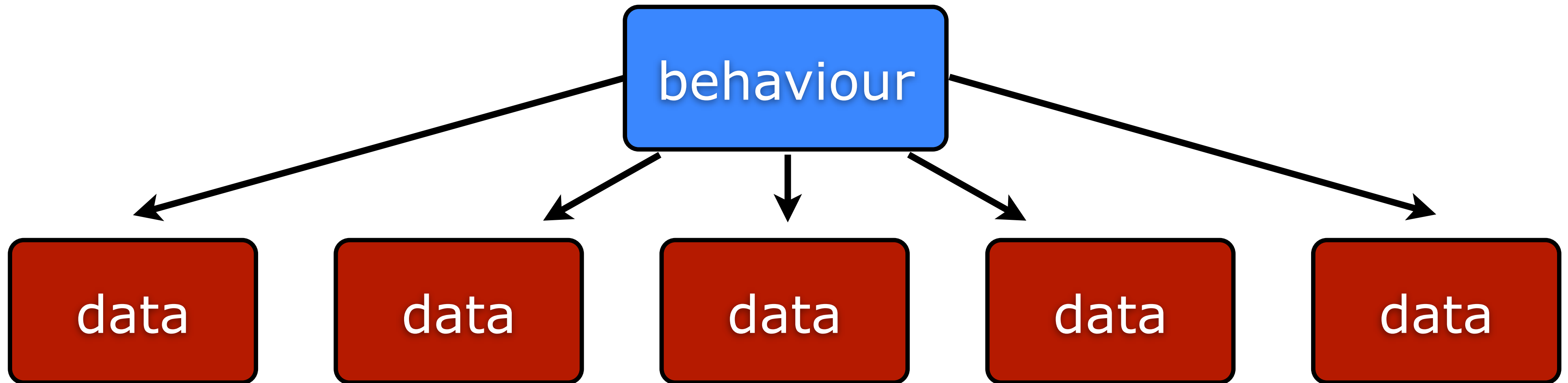
- one **interface** defining behaviour
- many **representations** (concrete classes)

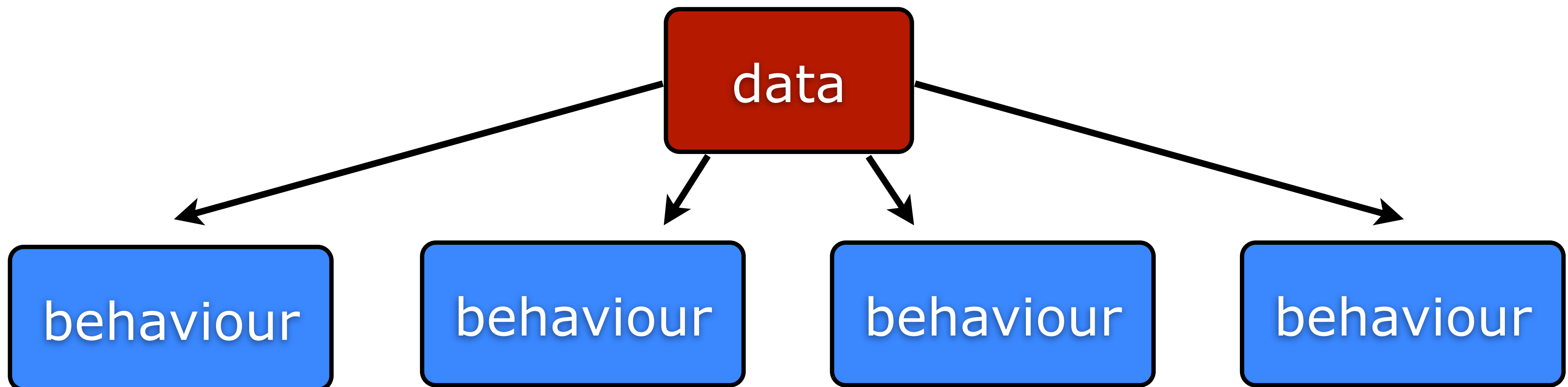
FP

- one underlying **representation** (data type)
- many **behavioural units** (functions)



OOP vs. FP





Traits

```
trait Comparable[T] {
  def compareTo(other: T): Int
  def lessThan(other: T): Boolean
  def equivalent(other: T): Boolean
  def greaterThan(other: T): Boolean
}
```

Traits

```
trait Comparable[T] {
  def compareTo(other: T): Int

  def lessThan(other: T): Boolean =
    compareTo(other) < 0
  def equivalent(other: T): Boolean =
    compareTo(other) == 0
  def greaterThan(other: T): Boolean =
    compareTo(other) > 0
}
```

Traits

```
case class Seconds(value: Int)
    extends Comparable[Seconds] {

  def compareTo(other: Seconds) =
    value - other.value
}

Seconds(1).lessThan(Seconds(2))     // true
Seconds(1).lessThan(Seconds(1))     // false
Seconds(1).greaterThan(Seconds(-2)) // true
```

Traits

```
trait Foo {
  def foo: String = "foo"
}

trait Bar {
  def bar: Int = 42
}

class Baz extends Foo with Bar

val baz = new Baz
print(baz.foo) // "foo"
print(baz.bar) // 42
```

Traits

```scala
trait Foo {
  def foo: String = "foo"
}

trait Bar {
  def bar: Int = 42
}

class Baz

val baz = new Baz with Foo with Bar
print(baz.foo) // "foo"
print(baz.bar) // 42
```

Lazy Evaluation

```
class Laziness {
  lazy val property = {
    println("Property has been initialized.")
    42
  }
}

val a = new Laziness
a.property
```

Strict Evaluation

```
def first[A,B](a: A, b: B): A = a

first(1, 1 / 0)
```

Lazy Evaluation

```
def first[A,B](a: A, b: => B): A = a

first(1, 1 / 0)
```

Lazy Evaluation

```
def double(b: => Int) = b + b

double({ println("Evaluating b..."); 2 })
```

Lazy Evaluation

```
def unless(cond: Boolean)(body: => Unit) = {
  if (!cond) body
}

unless (1 == 2) {
  println("1 != 2")
}
```

Logo Origin

Stairs from EPFL (École Polytechnique Fédérale de Lausanne), where Scala was born.

Thank You

Questions?