Scala

10 000 foot view and it’s FP features

Functional and logic programming

Aurel Paulovič May 2018

Scala language

- Heavily used in the industry (Twitter, LinkedIn, Apache Spark, Akka, …)
- Static strong typing
- Runs on JVM
  - Compiles to java bytecode
  - Interoperable with languages/libraries running on JVM
- *Almost* everything is an expression (i.e. has return value)
- Turing-complete type system

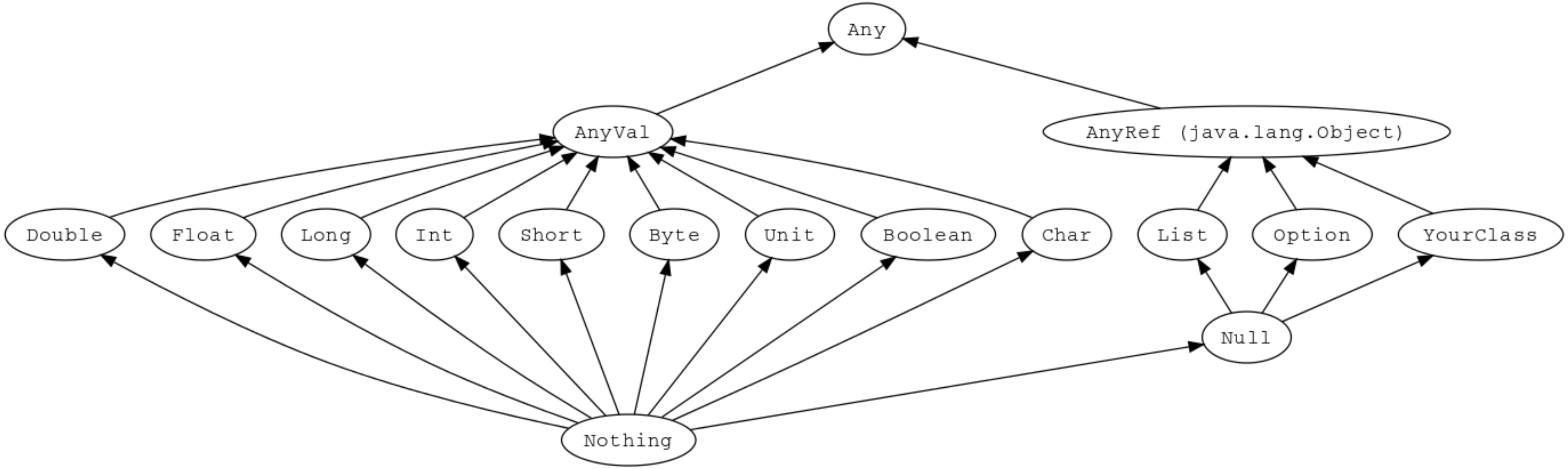
Scala syntax

- java-like
- `def` for method/function definition
- `func(name: Type, name2: Type2): Type3`
- Type inference
- No semicolons
- Sometimes can omit parentheses
- Sometimes can omit dots between `instance.method` invocation
- Lots of syntactic sugar
- No `return`

Scala OO features

- Everything is an object
- Classes
- Traits
- Singleton objects
- Inheritance polymorphism
- Method overloading
- Access modifiers
- Unified access
- Variance
  - E.g. is `List[Dog] <: List[Animal]`

Unified type hierarchy



OOP

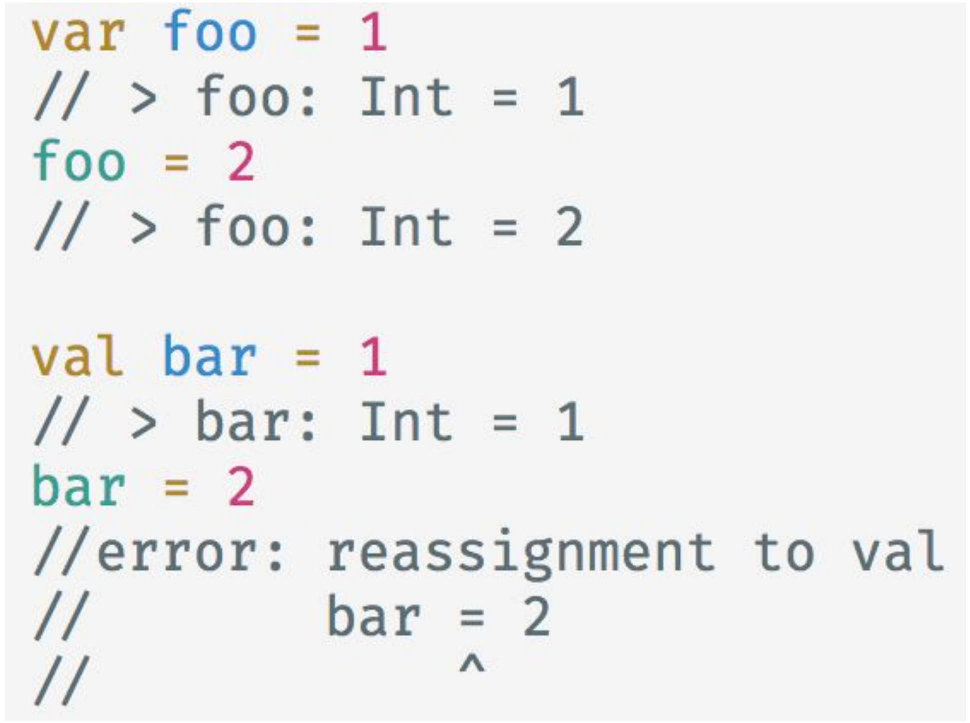
```scala
trait Animal {
  def makeSound: String
  def name: String
  def run(): Unit = println("running...")
}

class Dog (val name: String) extends Animal {
  def this(firstName: String, lastName: String) = this(s"$firstName, the $lastName")

  override def toString(): String = name

  override def makeSound = "woof"
}

val d1 = new Dog("Snuffles", "Snowball")
println(d1.toString)
println(d1.makeSound)
d1.run()
```

Scala FP features

- Immutable references (values)
- Expressions
- Higher-order functions
- Case classes
- Pattern matching
- FP-style core library APIs

Variables vs values

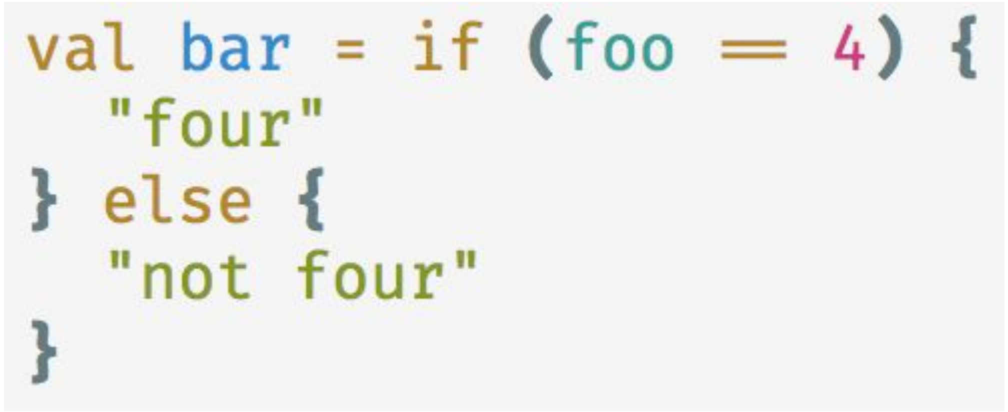
- `var` = variable
  - Multiple assignments
- `val` = value
  - Single assignment



```
var foo = 1
// > foo: Int = 1
foo = 2
// > foo: Int = 2

val bar = 1
// > bar: Int = 1
bar = 2
//error: reassignment to val
//         bar = 2
//             ^
```

Expressions

- Result into a value
- Have result type



```
val bar = if (foo == 4) {
  "four"
} else {
  "not four"
}
```

Higher-order functions

- Function can be assigned to a variable/value
- Function can take functions as arguments
- Function can return a function

Function assignment

```scala
val greeting = (name: String) => s"Hello, $name"
```

Function as an argument

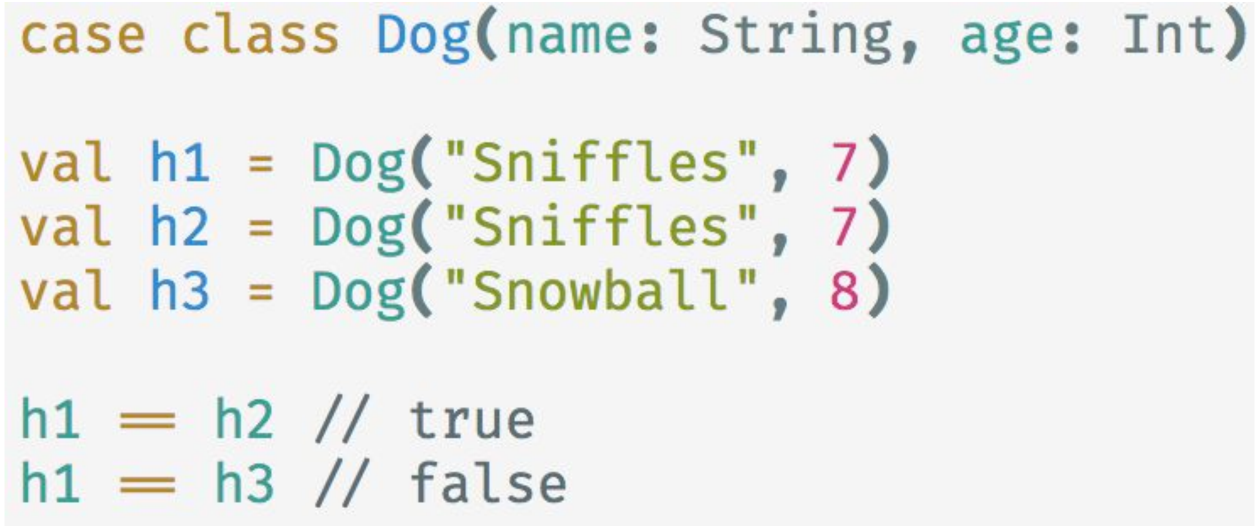
```
def doOperation[T, U](arg1: T, arg2: T, op: (T, T) ⇒ U): U = {
  op(arg1, arg2)
}

doOperation(1, 14, (a: Int, b: Int) ⇒ a + b) // 15
```

Function as a return value

```
def counter(start: Int) = {
  var current = start
  () ⇒ {
    val res = current
    current += 1
    res
  }
}

val c1 = counter(10)

c1() // 10
c1() // 11
c1() // 12
```

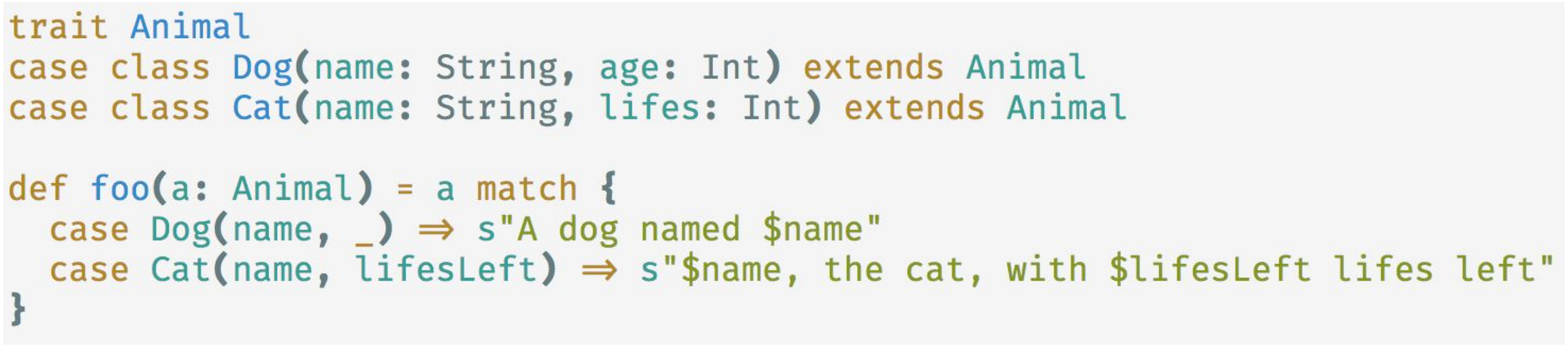
Case class

- Immutable
- Automatically generated `hashCode`, `equals`, extractors, ...
- Uses structural comparison



```scala
case class Dog(name: String, age: Int)

val h1 = Dog("Sniffles", 7)
val h2 = Dog("Sniffles", 7)
val h3 = Dog("Snowball", 8)

h1 == h2 // true
h1 == h3 // false
```

Pattern matching



```scala
trait Animal
case class Dog(name: String, age: Int) extends Animal
case class Cat(name: String, lifes: Int) extends Animal

def foo(a: Animal) = a match {
  case Dog(name, _) ⇒ s"A dog named $name"
  case Cat(name, lifesLeft) ⇒ s"$name, the cat, with $lifesLeft lifes left"
}
```

FP-style core library APIs

- Favour immutability (immutable collections, case classes, etc.)
- Higher-order functions
- Pure functions

```
trait Animal
case class Dog(name: String, age: Int) extends Animal
case class Cat(name: String, lifes: Int) extends Animal

def grantLife(cat: Cat): Cat = {
  cat.copy(lifes = cat.lifes + 1)
}

val animals = Seq(Dog("Snuffles", 7), Cat("Garfield", 4), Cat("Tom", 3))

animals.collect{ case x @ Cat(name, lives) ⇒ x }.map(grantLife).filter(_.lifes > 4)
// List(Cat(Garfield,5))
```

Other Scala features

- Implicit parameters
- Implicit conversions
- Generics (parametric polymorphism)
- ...

What makes Scala a FP language?

- Scala is a hybrid language
- You can
  - do OOP in Scala
  - write imperative code
  - have side-effects

1. Functions as first-class citizens (function as value, argument, return value)
2. FP-style core APIs
3. Idioms
4. Community & libraries

What makes Scala a FP language?

1. Functions as first-class citizens (function as value, argument, return value)

- Easy syntax for function definition, lambdas, etc.
- Function composition
- Currying
- ...

What makes Scala a FP language?

1. Functions as first-class citizens (function as value, argument, return value)
2. FP-style core APIs

- Favour immutability
- Pure functions - no side effects
- Higher-order functions

What makes Scala a FP language?

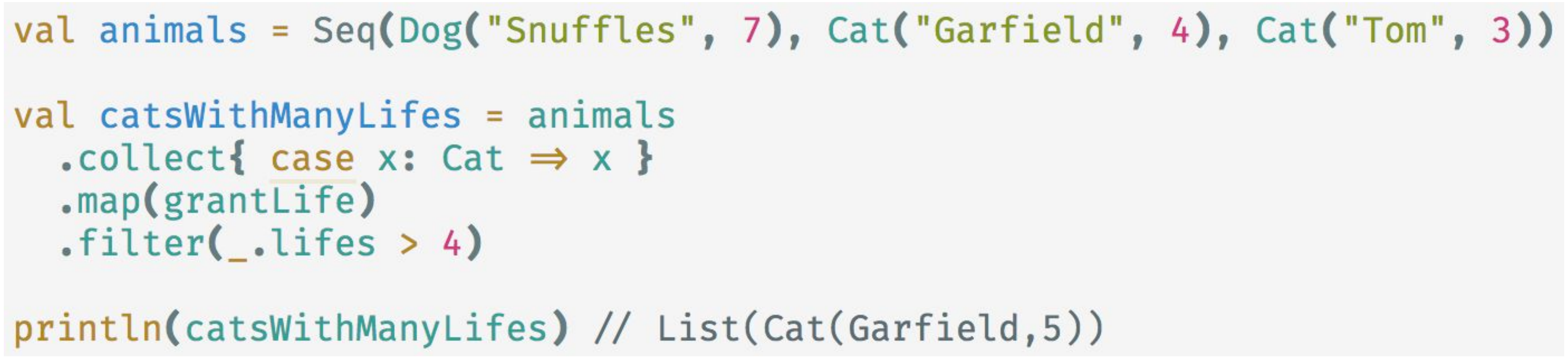
1. Functions as first-class citizens (function as value, argument, return value)
2. FP-style core APIs
3. Idioms

- Instead of imperative constructs prefer functional constructs

Idioms

```scala
val animals = Seq(Dog("Snuffles", 7), Cat("Garfield", 4), Cat("Tom", 3))

val animalsIterator = animals.iterator
val catsWithManyLifes = scala.collection.mutable.Buffer.empty[Cat]
while (animalsIterator.hasNext) {
  val animal = animalsIterator.next()
  if (animal.isInstanceOf[Cat]) {
    val upgradedCat = grantLife(animal.asInstanceOf[Cat])
    if (upgradedCat.lifes > 4) {
      catsWithManyLifes += upgradedCat
    }
  }
}

println(catsWithManyLifes) // ArrayBuffer(Cat(Garfield,5))
```

Idioms



```scala
val animals = Seq(Dog("Snuffles", 7), Cat("Garfield", 4), Cat("Tom", 3))

val catsWithManyLifes = animals
  .collect{ case x: Cat ⇒ x }
  .map(grantLife)
  .filter(_.lifes > 4)

println(catsWithManyLifes) // List(Cat(Garfield,5))
```

What makes Scala a FP language?

1. Functions as first-class citizens (function as value, argument, return value)
2. FP-style core APIs
3. Idioms
4. Community & libraries

- Libraries for FP (scalaz, cats, monix, …)
- Scala-idiomatic FP-style APIs
- Push for advanced FP concepts (monads, type-classes, recursion schemes, …)

Resources

- https://www.scala-lang.org
- https://github.com/lauris/awesome-scala
- https://www.manning.com/books/functional-programming-in-scala
- https://underscore.io/books/scala-with-cats/
- https://monix.io
- https://github.com/milessabin/shapeless