What’s the big deal with Scala?

praful@thoughtworks.com

Introduction to Scala

- Runs on JVM
- Fuses object-oriented & functional paradigms though biased towards functional style
- Statically-typed

As expressive as Ruby,
As performant as Java

Why am I comparing with Ruby and Java?

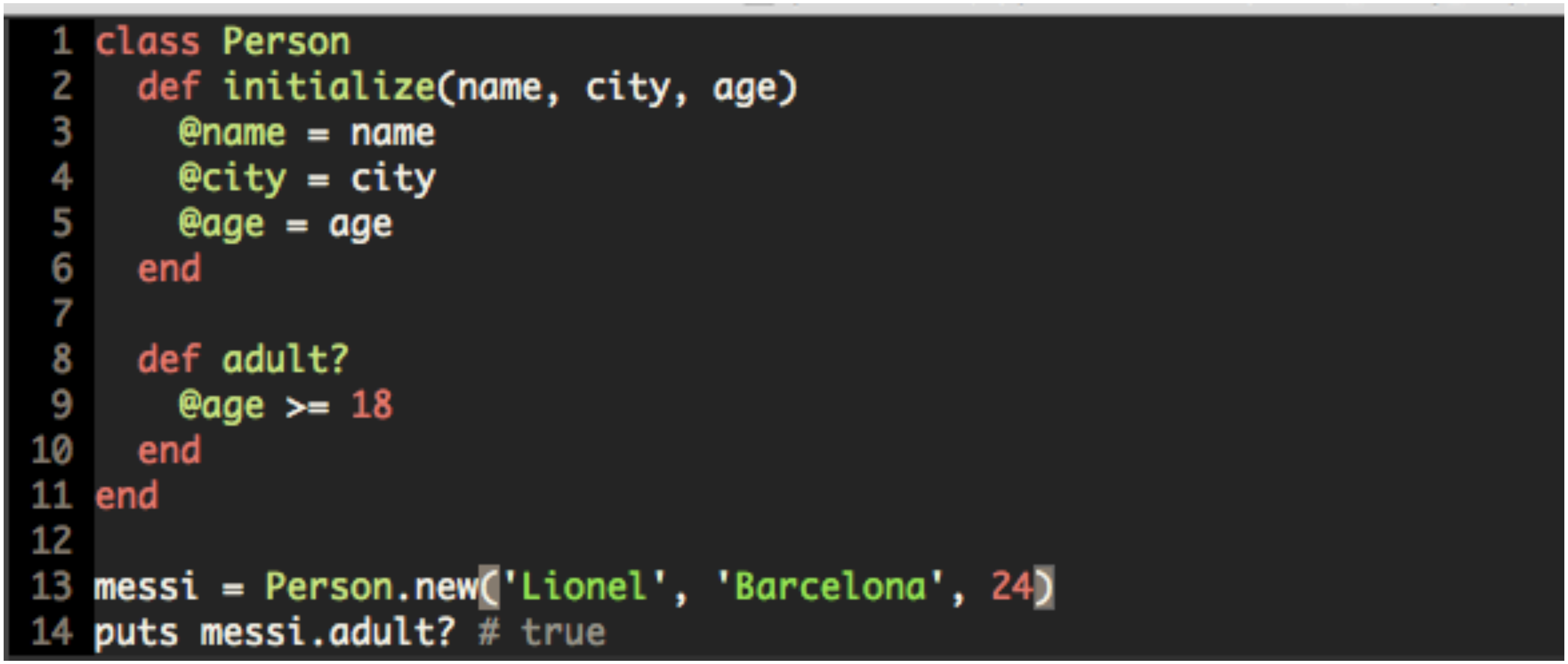
Disclaimer

- Professionally a Ruby and Java developer
- Haven’t worked with Scala on a “real” project
- Based on my experience working with the Credit Union Findr

Why Expressiveness?

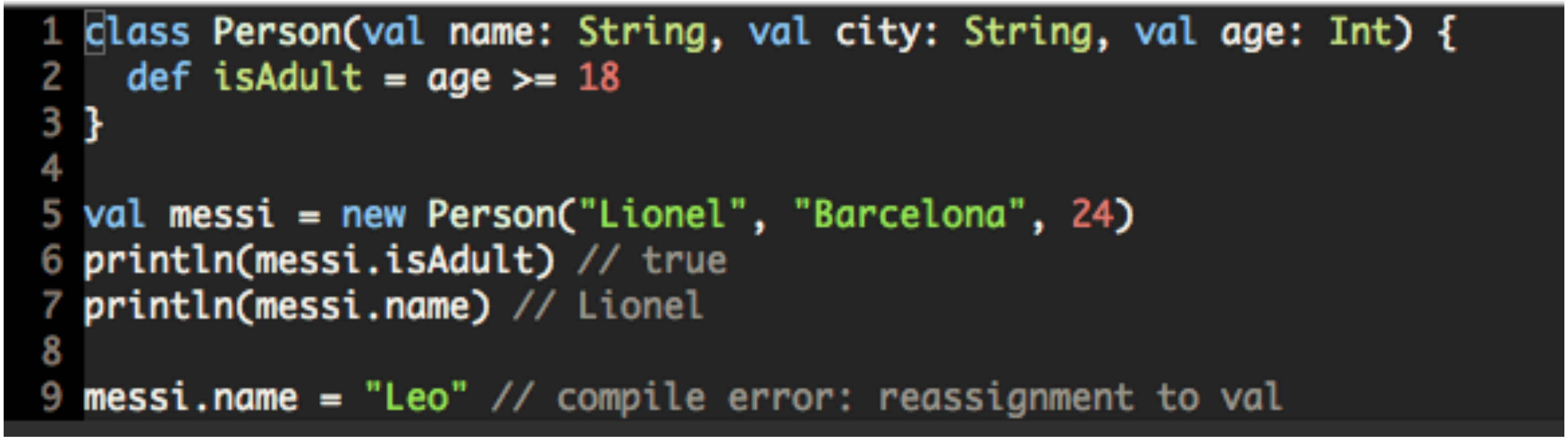
- Not just pretty
- Reduces boilerplate code
- Less code to deal with & maintain
- Elegant solution

Ruby class definition



```ruby
class Person
  def initialize(name, city, age)
    @name = name
    @city = city
    @age = age
  end

  def adult?
    @age >= 18
  end
end

messi = Person.new('Lionel', 'Barcelona', 24)
puts messi.adult? # true
```

Scala class definition

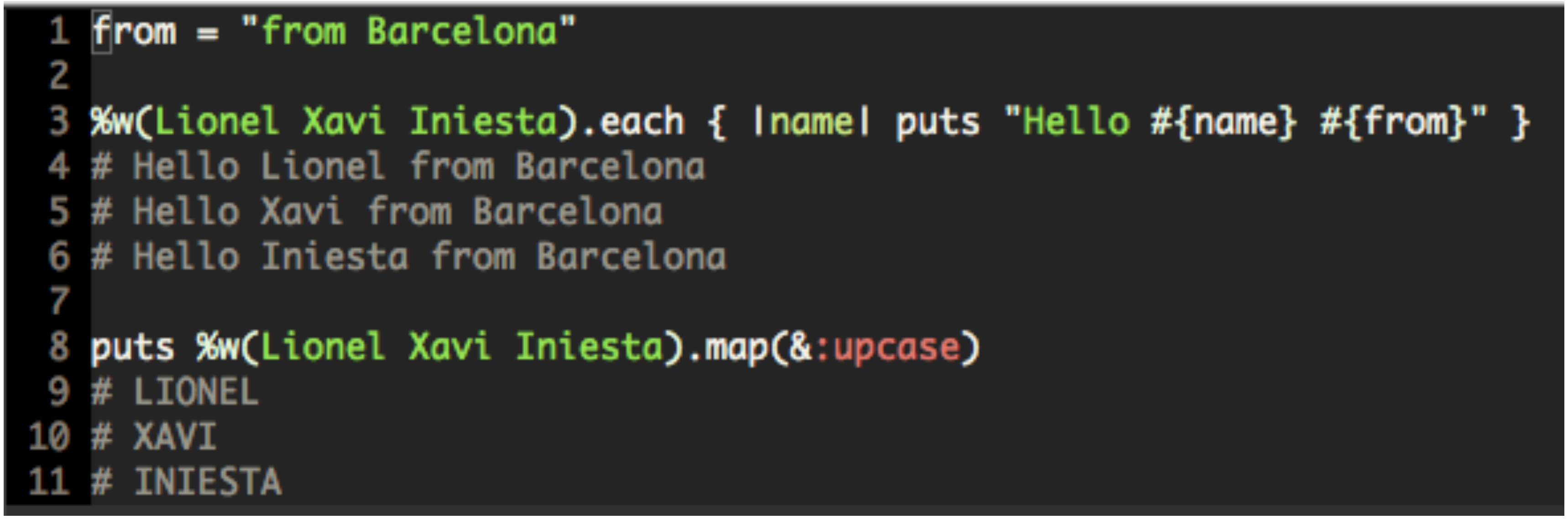


```scala
class Person(val name: String, val city: String, val age: Int) {
  def isAdult = age >= 18
}

val messi = new Person("Lionel", "Barcelona", 24)
println(messi.isAdult) // true
println(messi.name) // Lionel

messi.name = "Leo" // compile error: reassignment to val
```

Closures

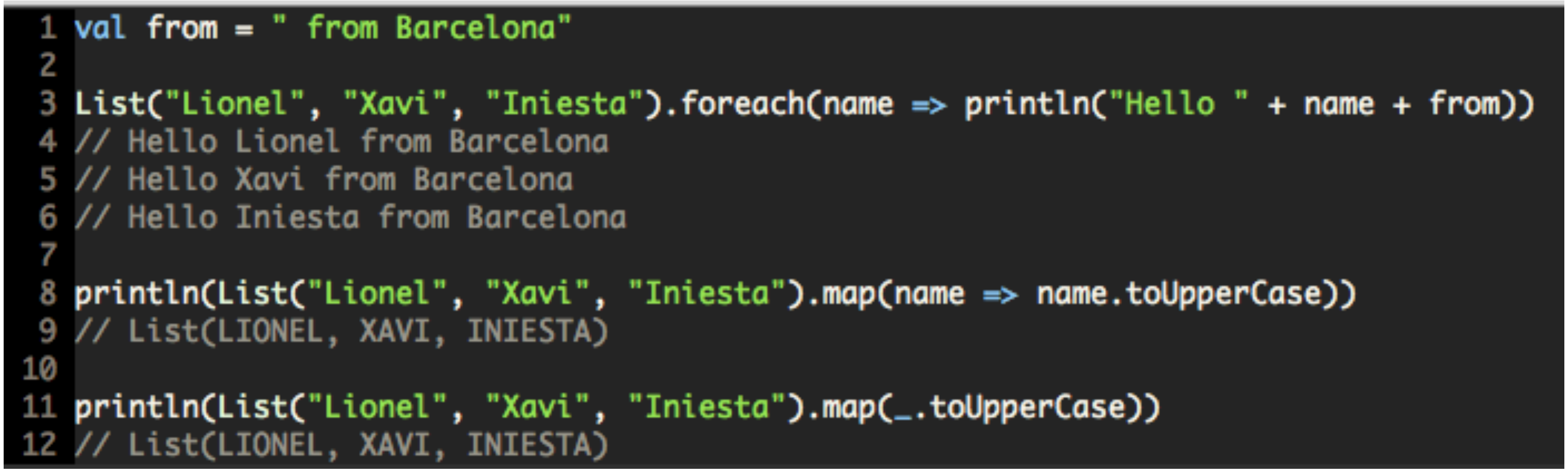
- Closures are blocks of code that can be passed around as parameters
- Closures “close over” variables outside of its scope

Closures in Ruby



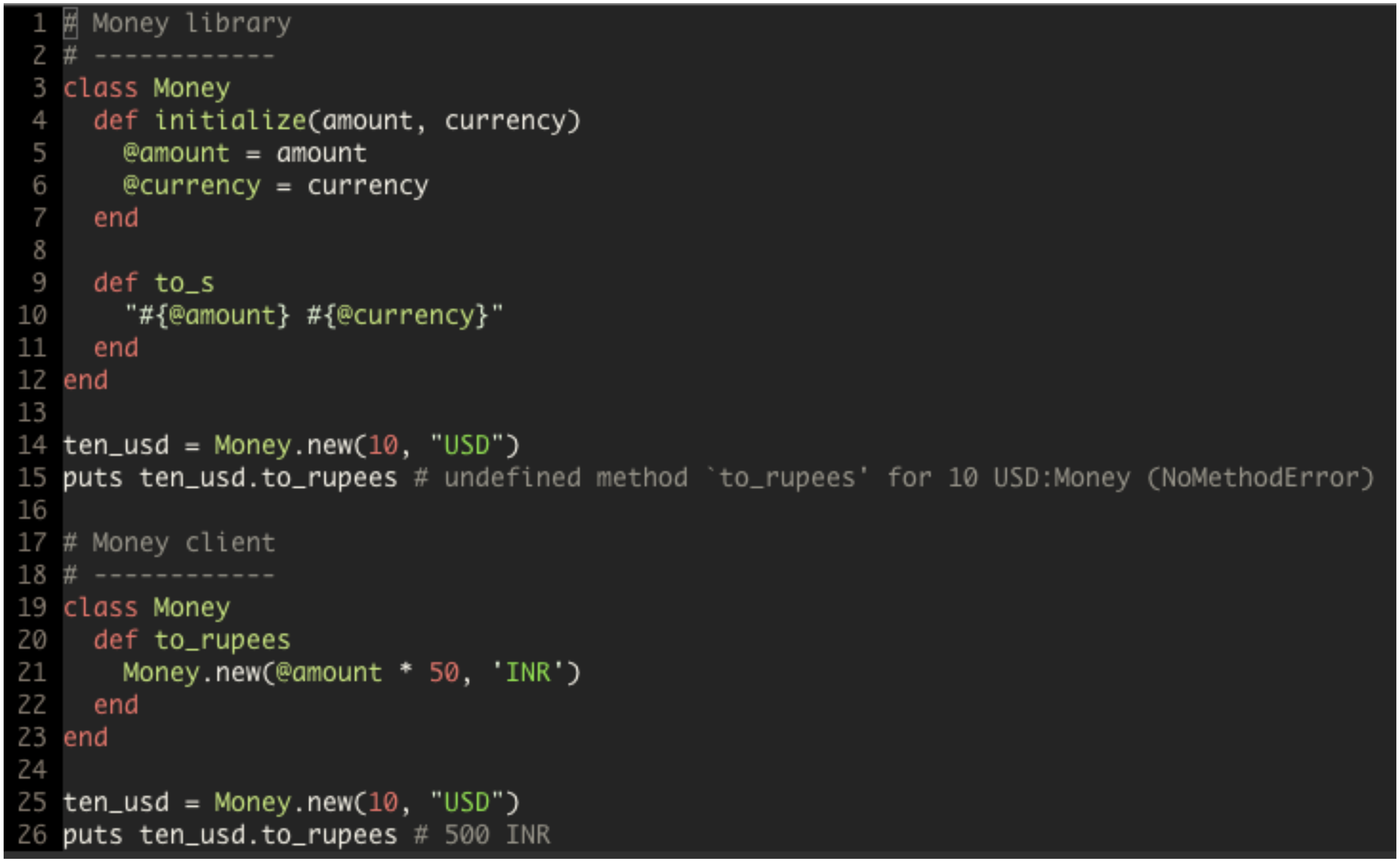
```ruby
from = "from Barcelona"

%w(Lionel Xavi Iniesta).each { |name| puts "Hello #{name} #{from}" }
# Hello Lionel from Barcelona
# Hello Xavi from Barcelona
# Hello Iniesta from Barcelona

puts %w(Lionel Xavi Iniesta).map(&:upcase)
# LIONEL
# XAVI
# INIESTA
```

Closures in Scala



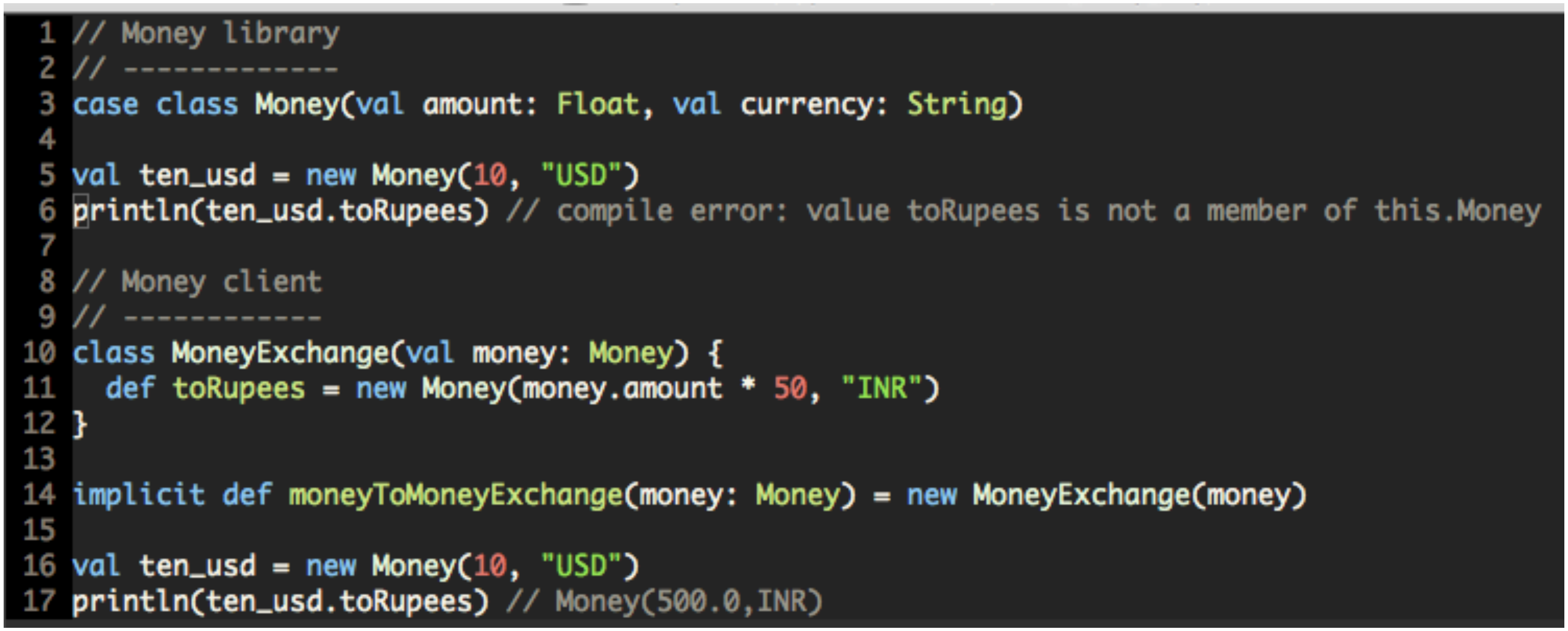
```scala
val from = " from Barcelona"

List("Lionel", "Xavi", "Iniesta").foreach(name => println("Hello " + name + from))
// Hello Lionel from Barcelona
// Hello Xavi from Barcelona
// Hello Iniesta from Barcelona

println(List("Lionel", "Xavi", "Iniesta").map(name => name.toUpperCase))
// List(LIONEL, XAVI, INIESTA)

println(List("Lionel", "Xavi", "Iniesta").map(_.toUpperCase))
// List(LIONEL, XAVI, INIESTA)
```

Ruby monkey patching



```ruby
# Money library
# ------------
class Money
  def initialize(amount, currency)
    @amount = amount
    @currency = currency
  end

  def to_s
    "#{@amount} #{@currency}"
  end
end

ten_usd = Money.new(10, "USD")
puts ten_usd.to_rupees # undefined method `to_rupees' for 10 USD:Money (NoMethodError)

# Money client
# ------------
class Money
  def to_rupees
    Money.new(@amount * 50, 'INR')
  end
end

ten_usd = Money.new(10, "USD")
puts ten_usd.to_rupees # 500 INR
```

Scala implicit conversions



```scala
// Money library
// -------------
case class Money(val amount: Float, val currency: String)

val ten_usd = new Money(10, "USD")
println(ten_usd.toRupees) // compile error: value toRupees is not a member of this.Money

// Money client
// ------------
class MoneyExchange(val money: Money) {
  def toRupees = new Money(money.amount * 50, "INR")
}

implicit def moneyToMoneyExchange(money: Money) = new MoneyExchange(money)

val ten_usd = new Money(10, "USD")
println(ten_usd.toRupees) // Money(500.0,INR)
```

Performance and Scalability constructs

Tail recursion

- Special type of function recursion
- Final action taken in a function is the recursive call
- Can avoid penalty of creating a new stack frame for each recursive call

Lets look at some code...

Tail recursion

Parallel collections

- Just like regular collections
- Can be operated upon by multiple cores
- Use Divide-and-conquer algorithm
- Write-your-own

Lets look at some code...

Parallel collections

Actors

- Actors are high-level concurrency construct as opposed to threads and shared memory model
- Actors communicate via message passing
- Pattern matching is used for message processing

Lets look at some code...

Actors

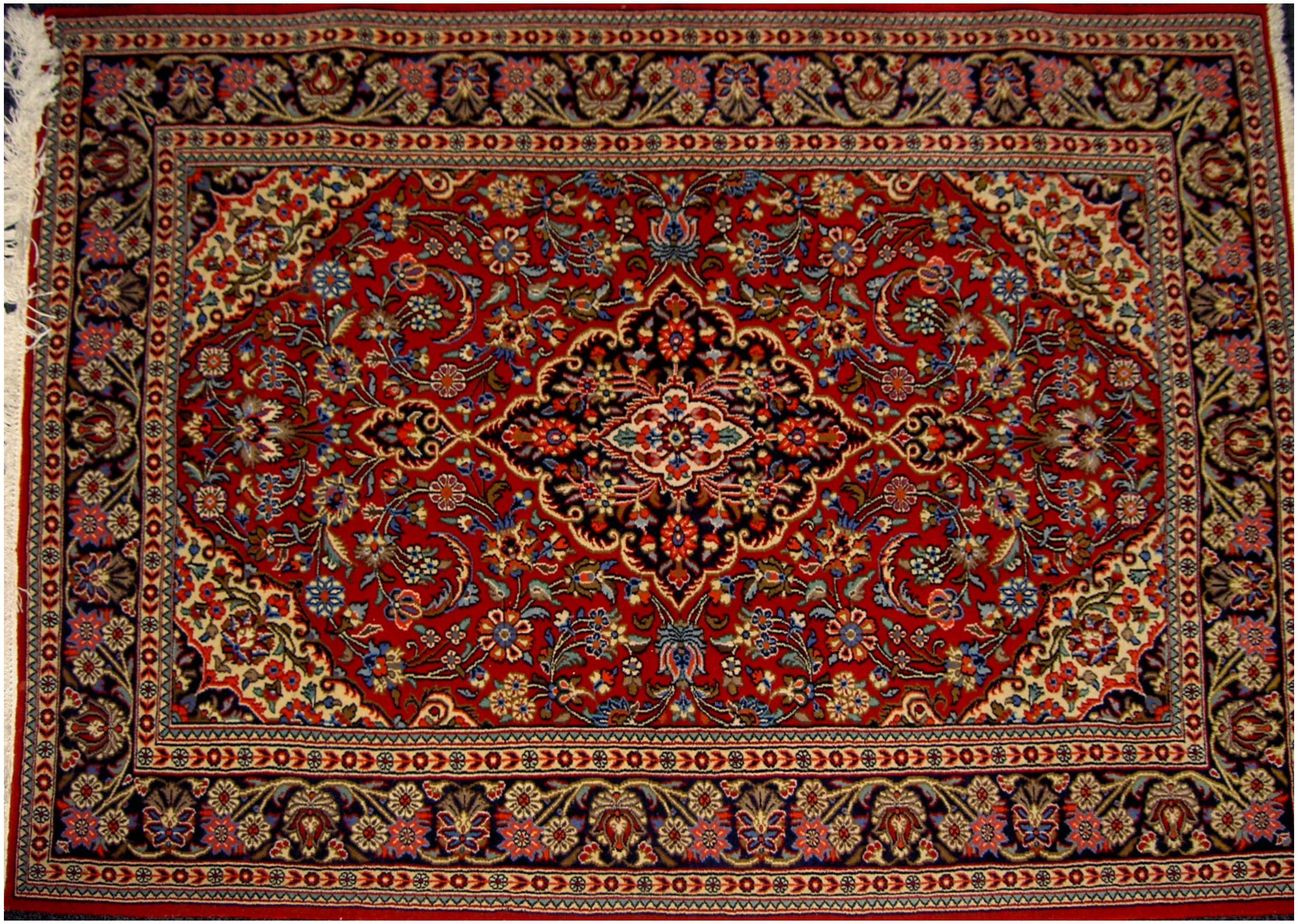
Criticism

- Slow compiler
  - sbt incremental compilation
  - Daemon compiler
- Too many features

Other items of interest

- REPL
- TypeSafe (Scala, Akka, Play!)
- Heroku

Functional yet beautiful

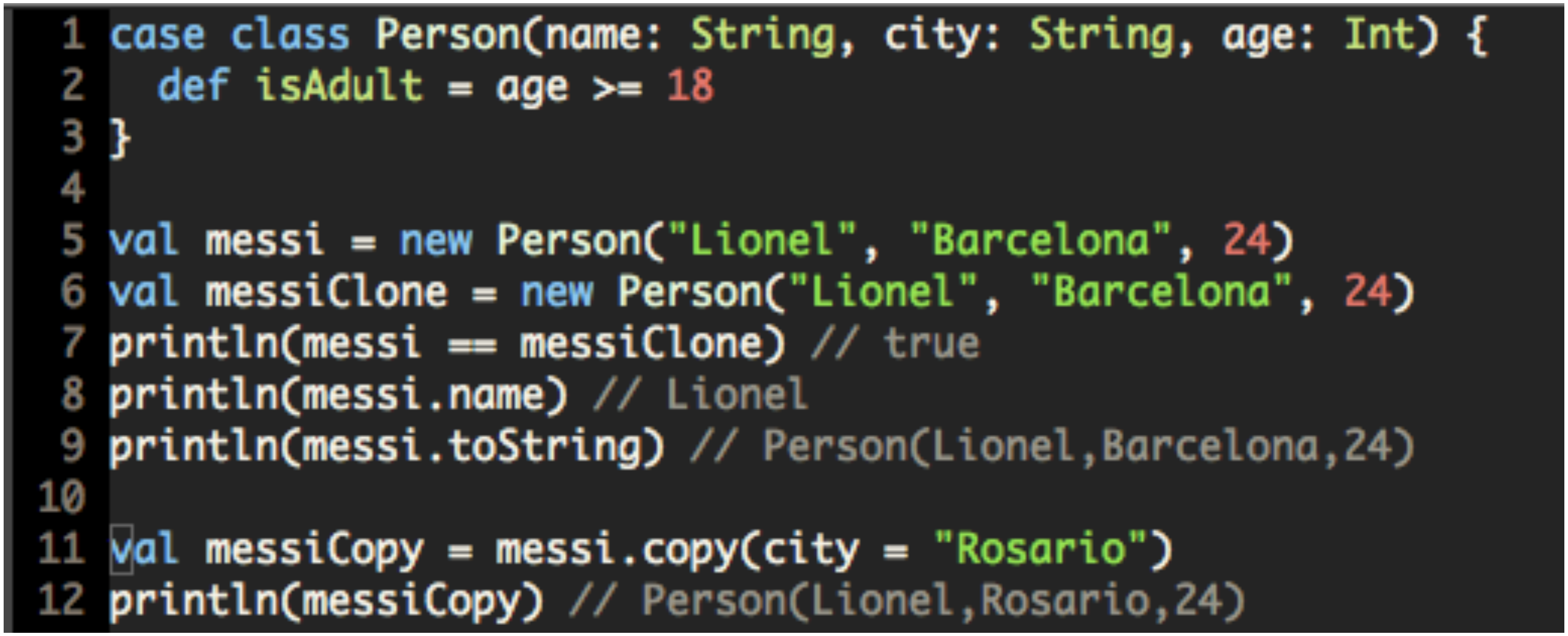


Rug from Qom, Iran http://upload.wikimedia.org/wikipedia/commons/c/ca/Farsh_Qom.JPG)

Thank you!

@todkar

Scala case class definition

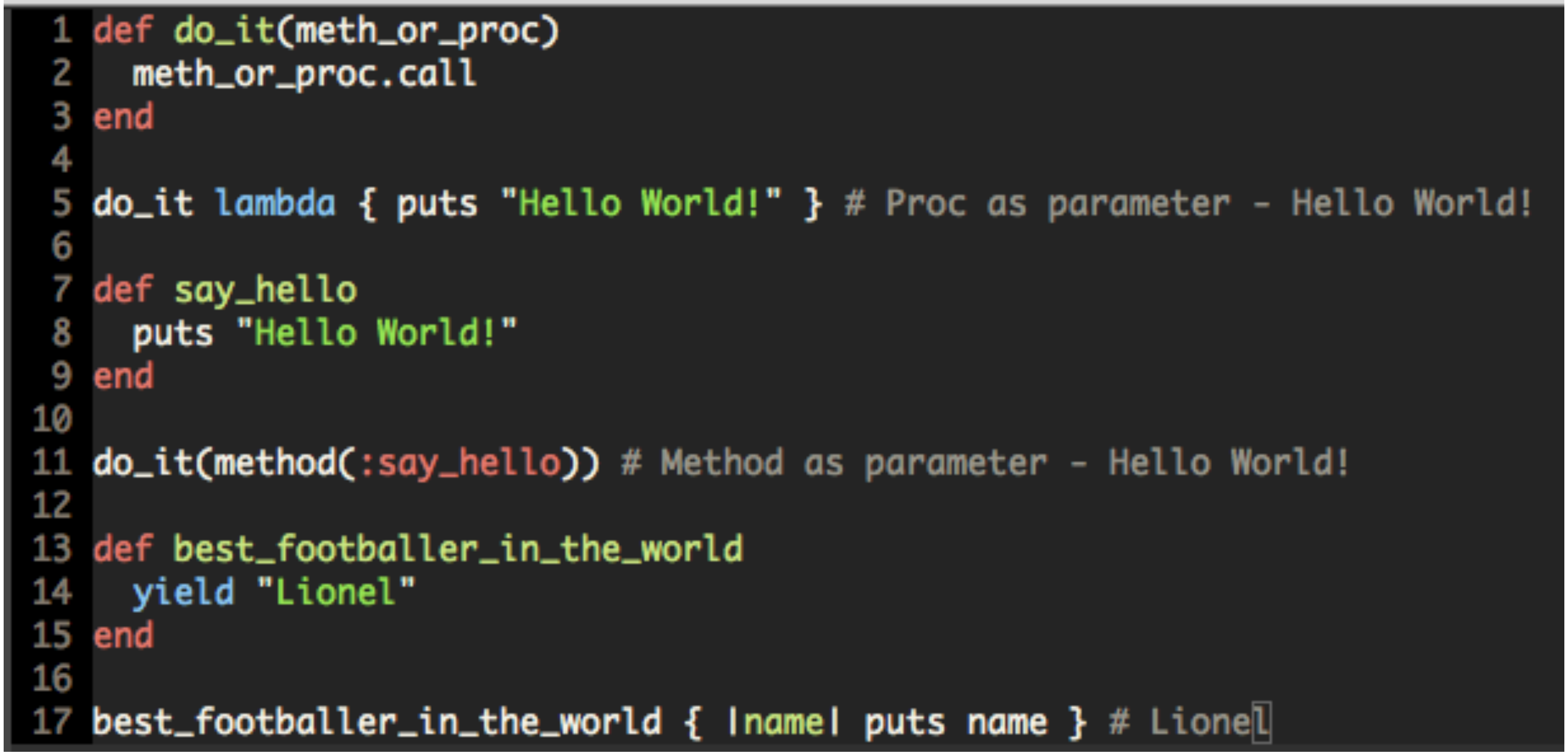


```scala
case class Person(name: String, city: String, age: Int) {
  def isAdult = age >= 18
}

val messi = new Person("Lionel", "Barcelona", 24)
val messiClone = new Person("Lionel", "Barcelona", 24)
println(messi == messiClone) // true
println(messi.name) // Lionel
println(messi.toString) // Person(Lionel,Barcelona,24)

val messiCopy = messi.copy(city = "Rosario")
println(messiCopy) // Person(Lionel,Rosario,24)
```

Higher-Order Functions

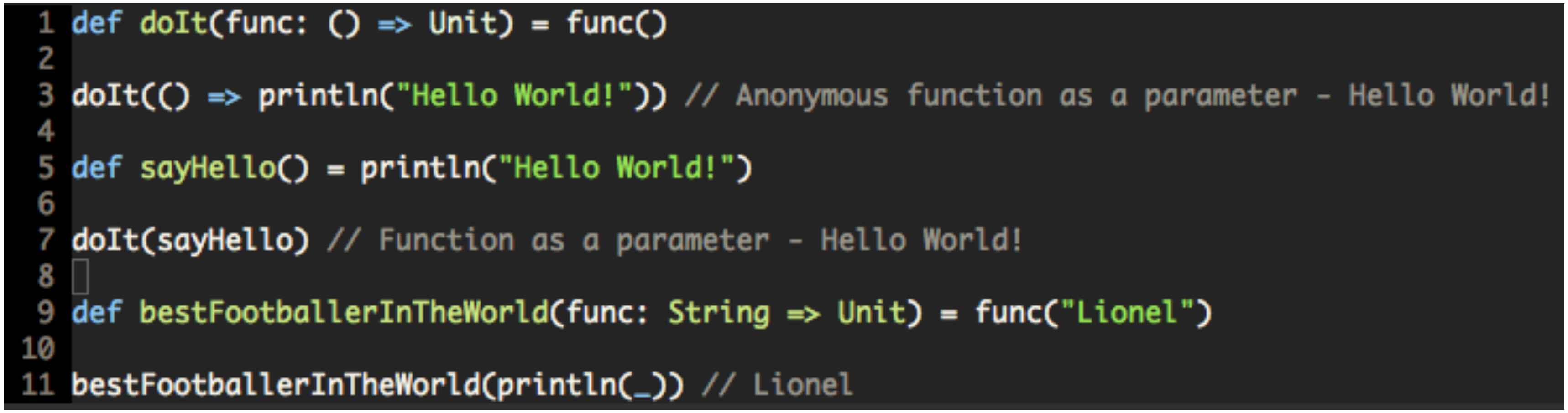
- Higher-Order Functions are functions that either input or output functions.

Higher-Order Functions in Ruby



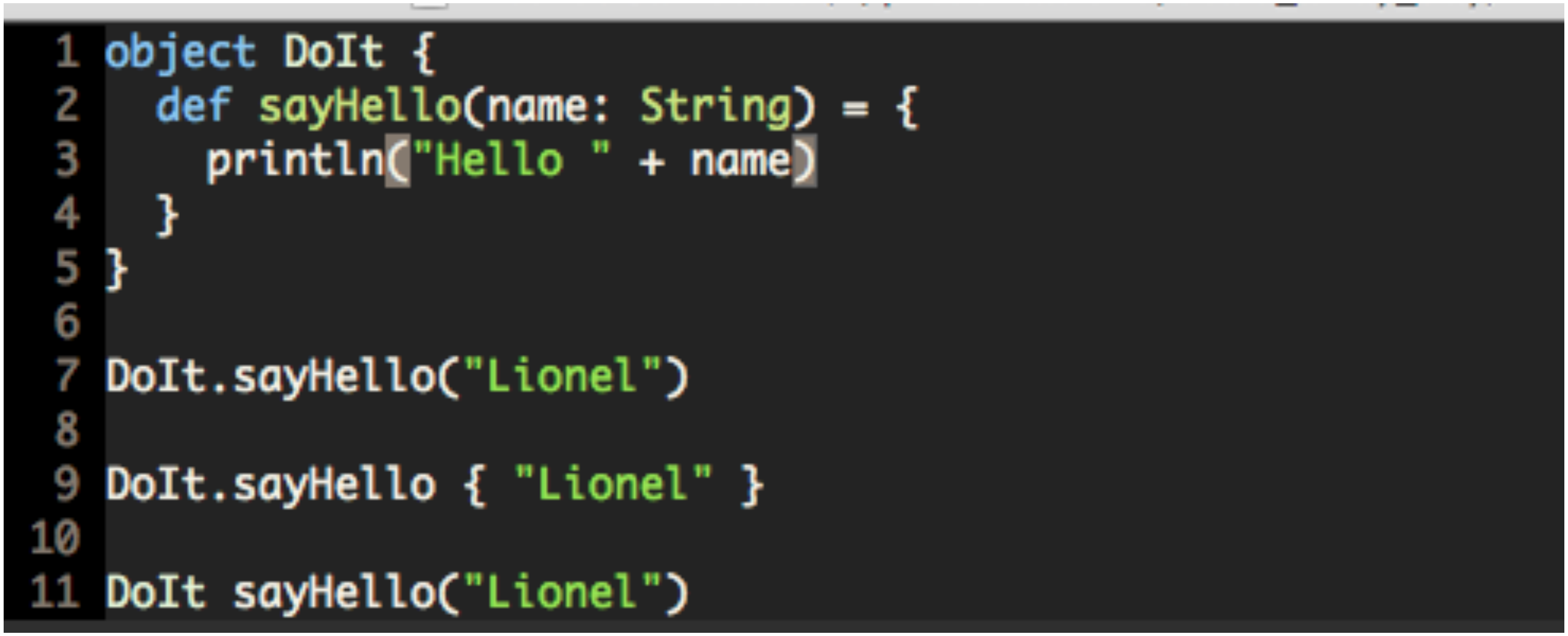
```ruby
def do_it(meth_or_proc)
  meth_or_proc.call
end

do_it lambda { puts "Hello World!" } # Proc as parameter - Hello World!

def say_hello
  puts "Hello World!"
end

do_it(method(:say_hello)) # Method as parameter - Hello World!

def best_footballer_in_the_world
  yield "Lionel"
end

best_footballer_in_the_world { |name| puts name } # Lionel
```

Higher-Order functions in Scala



```scala
def doIt(func: () => Unit) = func()

doIt(() => println("Hello World!")) // Anonymous function as a parameter - Hello World!

def sayHello() = println("Hello World!")

doIt(sayHello) // Function as a parameter - Hello World!

def bestFootballerInTheWorld(func: String => Unit) = func("Lionel")

bestFootballerInTheWorld(println(_)) // Lionel
```

Criticism



```
object DoIt {
  def sayHello(name: String) = {
    println("Hello " + name)
  }
}

DoIt.sayHello("Lionel")

DoIt.sayHello { "Lionel" }

DoIt sayHello("Lionel")
```