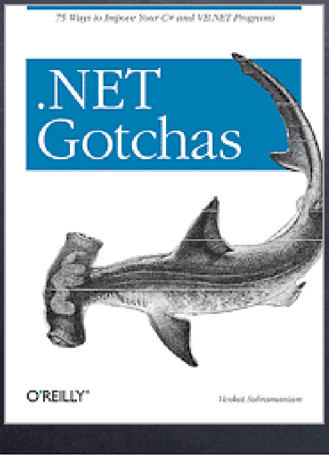
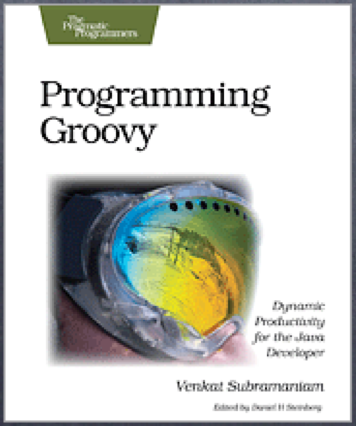
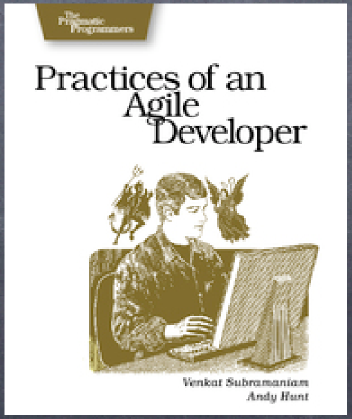
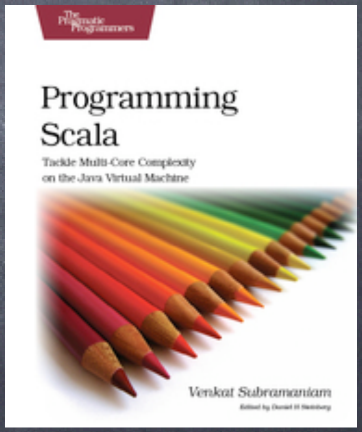
Programming Scala

Venkat Subramaniam
venkats@AgileDeveloper.com




The Pragmatic Programmers
Programming Groovy
Dynamic Productivity for the Java Developer
Venkat Subramaniam
Edited by Daniel H Steinberg




The Pragmatic Programmers
Programming Scala
Tackle Multi-Core Complexity on the Java Virtual Machine
Venkat Subramaniam
Edited by Daniel H Steinberg

Not Your Father’s Environment

Your World: Multiprocessors are Common Place

Multithreading on Steroids

“Well Written” Programs may be ill-fated on Multiprocessors

Cry for Higher Level Of Abstraction

Threading Support of Java/.NET Won't Cut It

As soon as you create a thread,
you worry how to control it

`synchronize` is latin for waste Concurrency

How can Functional Programming Help?

Assignment-less Programming

Immutable State

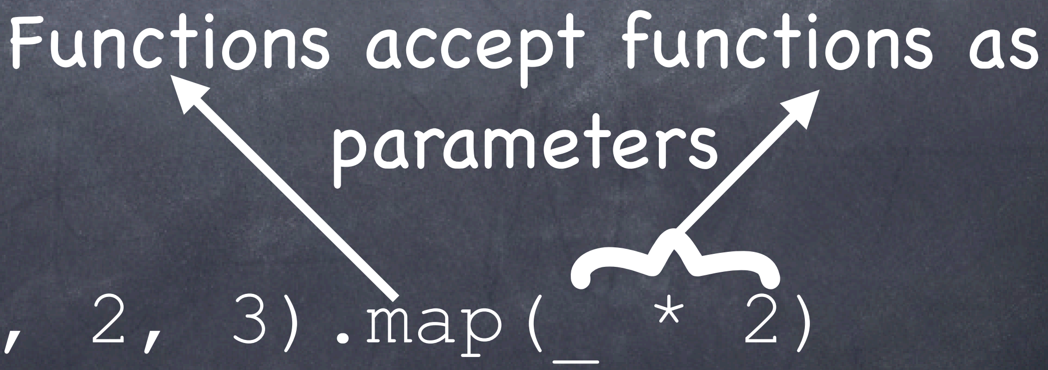
You can't Screwup what
you can't change

But What's FP?

Functions are first-class citizens

create them wherever you like,
store them, pass them around, …

Higher Order Functions



Functions accept functions as parameters

```
List(1, 2, 3).map(_ * 2)
```

What's Scala?

Old wine in a new bottle

Provides FP on the JVM

It's more of a cocktail

```
var total = 0
for(i <- 1 to 3)
  total += i
```

Supports Imperative (how to do) and Functional (what to do) style of coding

```
(1 to 3).foldLeft(0) {(v, e) => v + e}
(1 to 3).foldLeft(0) { _ + _ }
(0 /: (1 to 3)) { _ + _ }
```

What can it do for you?

event-based concurrency model

purely OO

intermixes well with Java

sensible static typing

concise

built on small kernel

highly scalable

Essence vs. Ceremony

```
public class HelloWorld
{
  public static void main(String[] args)
  {
    System.out.println("Hello World!");
  }
}
```

Why?

```
println("Hello World!")
```

; . () optional

```
for(i <- 1.to(3)) print("ho ")

for(i <- 1 to 3) print("ho ")
```

No Operators, but Supports Operator Overloading!

No Operators...

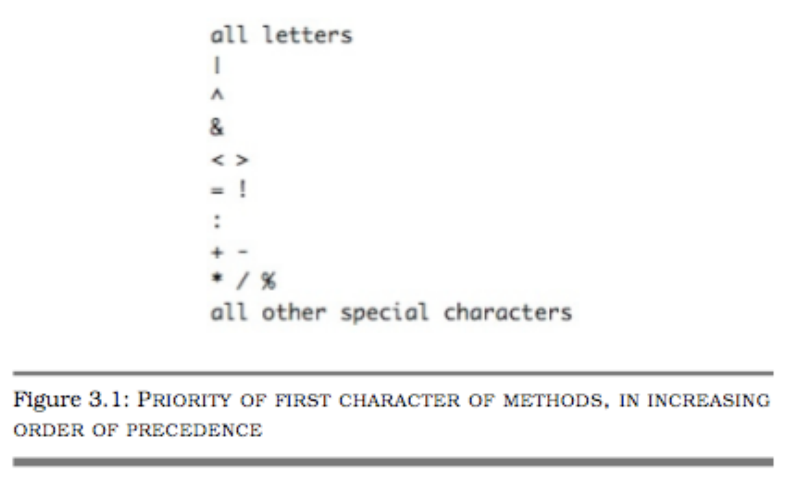
`a + b` is really `a.+(b)`

`+()` is simply a method

But, what about precedence?

first char of method name decides that!

Precedence



```
all letters
|
^
&
< >
= !
:
+ -
* / %
all other special characters
```

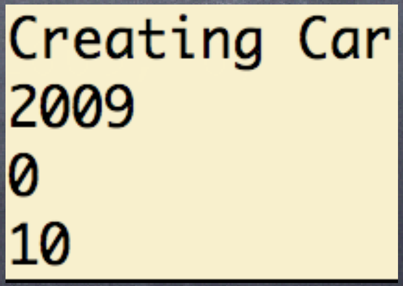
Figure 3.1: PRIORITY OF FIRST CHARACTER OF METHODS, IN INCREASING ORDER OF PRECEDENCE

```
class Sample
{
  def +(other: Sample) : Sample =
   { println("+ called"); this }
  def *(other: Sample) : Sample =
   { println("* called"); this }
}

val sample = new Sample
sample + sample * sample
```

```
* called
+ called
```

Cute Classes

```
class Car(val year: Int, var miles: Int)
{
  // what you put here goes into primary constructor
  println("Creating Car")

  def drive(dist: Int)
  {
    miles += dist
  }
}

val car = new Car(2009, 0)
println(car.year)
println(car.miles)
car drive 10
println(car.miles)
```



```
Creating Car
2009
0
10
```

Pure OO—No `static`

Everythin’s an Object

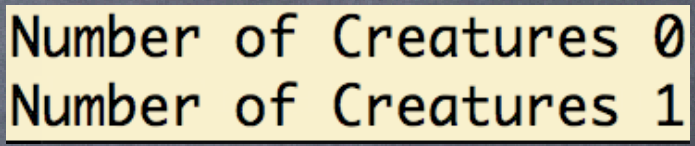
- For performance Int maps to Java primitive int

Has no support for `static`

- Something better!-Companion Objects

Companion Object

```
class Creature
{
  Creature.count += 1
}

object Creature
{
  var count: Int = 0
}

println("Number of Creatures " + Creature.count)
new Creature
println("Number of Creatures " + Creature.count)
```



```
Number of Creatures 0
Number of Creatures 1
```

vals and vars

- vars are variables
  - You can reassign to them
- vals provide immutability—they're valuables?!
  - Constant

```
var str1 : String = "hello"
val str2 : String = "hello"

str1 = "hi" // ok
str2 = "hi" // ERROR
```

Type Inference

```
var str = "hello"
def foo() = 2

// Scala knows str is String and
// foo returns Int

str = "hi" // OK

str = 4 // type-mismatch ERROR
```

Static typing that Works

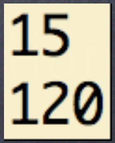
```
val nums = Array(1, 2, 3)

var objs = new Array[Object](4)

objs = nums // type-mismatch ERROR
            // (unlike Java)
```

Closures

- Function-values (code blocks) can bind to variables other than parameters and local variables

- These variables have to be closed before method invocation—hence closure

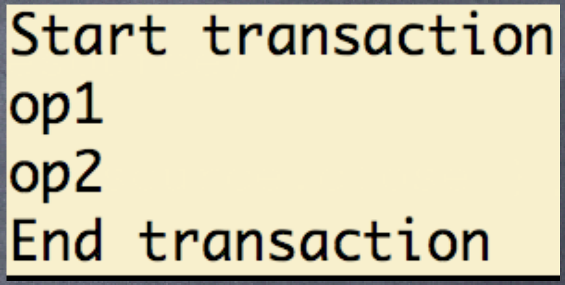
```
var total = 0
(1 to 5).foreach { total += _ }
println(total)

var product = 1
(1 to 5).foreach { product *= _ }
println(product)
```



```
15
120
```

Execute Around Method

```
class Resource
{
  println("Start transaction")

  def close() { println("End transaction") }
  def op1() { println("op1") }
  def op2() { println("op2") }
}
object Resource
{
  def use(block : Resource => Unit)
  {
    val resource = new Resource
    try {
      block(resource)
    }
    finally { resource.close }
  }
}
```



```
Start transaction
op1
op2
End transaction
```

```
Resource.use { resource =>
  resource.op1
  resource.op2
}
```

Traits—Cross Cutting Concerns

```
class Human(val name: String)
{
  def listen =
    println("I'm " + name + " your friend. I'm
listening...")
}

class Man(override val name: String) extends Human(name)

val sam = new Man("Sam")
sam.listen

//Friend is not modeled well
//Not clear
//Hard to reuse
```

Traits can help here

Think of them as interfaces with partial implementations

Traits—Cross Cutting Concerns

```
trait Friend
{
  val name : String //abstract
  def listen =
    println("I'm " + name + " your friend. I'm listening...")
}

class Human(val name: String)

class Man(override val name: String)
  extends Human(name)
  with Friend

class Dog(val name: String) extends Friend
{
  override def listen =
    println("Your friend " + name + " listening...")
}

def help(friend: Friend) { friend.listen }

help(new Man("Sam"))
help(new Dog("Casper"))
```

Traits—Cross Cutting Concerns

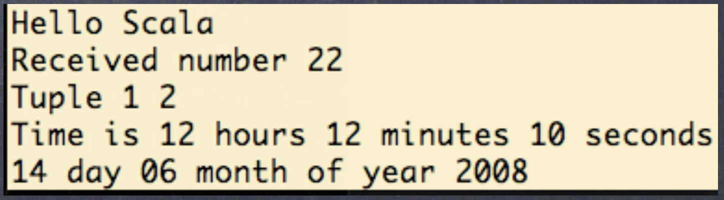
Not just at class level

```
class Cat(val name: String)

help(new Cat("Sally") with Friend)
```

Pattern Matching

- Quite powerful–here's a sample

```
def process(input : Any)
{
  val time = """(\d\d):(\d\d):(\d\d)""".r
  val date = """(\d\d)/(\d\d)/(\d\d\d\d)""".r

  input match {
    case "Scala" => println("Hello Scala")
    case (a, b) => println("Tuple " + a + " " + b)
    case num : Int => println("Received number " + num)
    case time(h, m, s) => printf("Time is %s hours %s minutes %s seconds\n", h, m, s)
    case date(m, d, y) => printf("%s day %s month of year %s\n", d, m, y)
  }
}

process("Scala")
process(22)
process(1, 2)
process("12:12:10")
process("06/14/2008")
```



```
Hello Scala
Received number 22
Tuple 1 2
Time is 12 hours 12 minutes 10 seconds
14 day 06 month of year 2008
```

Concurrency

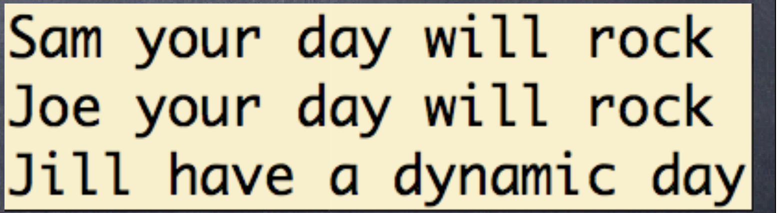
- No need for synchronized, wait, notify, ...
- Just create actors
- Send messages
- Make sure messages are immutable
- You’re done

Actor Based

```
import scala.actors.Actor._
import scala.actors.Actor

def getFortune() =
{
  val fortunes = List("your day will rock",
    "your day is filled with ceremony",
    "have a dynamic day",
    "keep smiling")

  fortunes((Math.random * 100).toInt % 4)
}

val fortuneTeller = actor {
  var condition = true
  while(condition)
  {
    receive {
      case "done" => condition = false
      case name : String =>
        sender ! name + " " + getFortune()
    }
  }
}
```

```
fortuneTeller ! "Sam"
fortuneTeller ! "Joe"
fortuneTeller ! "Jill"
fortuneTeller ! "done"

for(i <- 1 to 3)
{
  receive {
    case msg =>
      println(msg)
  }
}
```

Run’s in own thread



```
Sam your day will rock
Joe your day will rock
Jill have a dynamic day
```

Send message using !

`receive` to get msg

Thread Pooling

Each actor by default get's own thread

- Not efficient when large number of actors

- `react` can help
    - relinquishes thread while wait
    - gets a thread from pool when active
    - `react` never returns
        - so call tail recursive
        - or use `loop()`

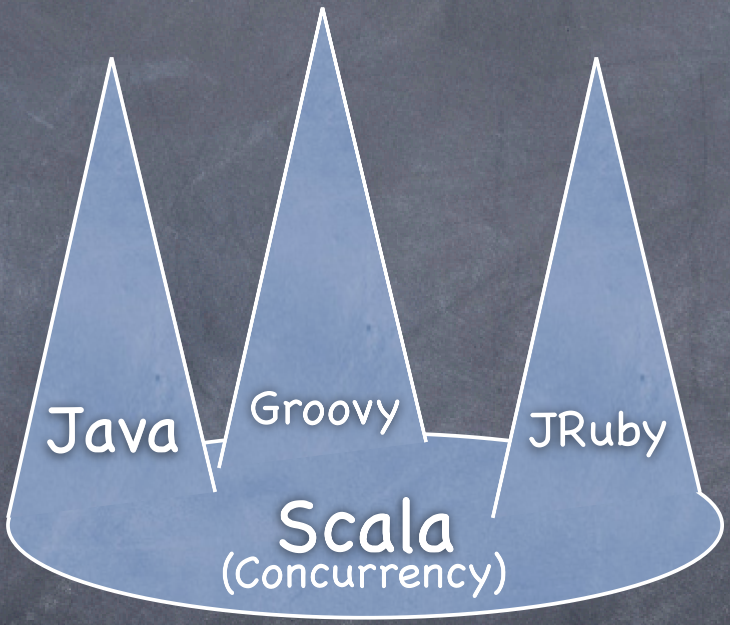
Using `react`

```
import scala.actors.Actor._

def info(msg: String)
{
  println(msg +
     " received by " +
        Thread.currentThread)
}

def useReceive()
{
  while(true)
  {
    receive { case msg : String => info(msg) }
  }
}

def useReact()
{
    react {
      case msg : String =>
        info(msg)
        useReact()
    }
}
```

```
hello1 received by Thread[Thread-5,5,main]
hello2 received by Thread[Thread-6,5,main]
hello3 received by Thread[Thread-6,5,main]
hello0 received by Thread[Thread-3,5,main]
hello1 received by Thread[Thread-5,5,main]
hello2 received by Thread[Thread-4,5,main]
hello3 received by Thread[Thread-4,5,main]
hello0 received by Thread[Thread-3,5,main]
hello1 received by Thread[Thread-5,5,main]
hello2 received by Thread[Thread-6,5,main]
hello3 received by Thread[Thread-4,5,main]
hello0 received by Thread[Thread-3,5,main]
```

```
val actors = List(actor { useReceive },
  actor { useReceive },
  actor { useReact },
  actor {useReact})

for(i <- 1 to 12)
{
  actors(i % 4) ! "hello" + (i % 4)
  Thread.sleep(1000)
}
```

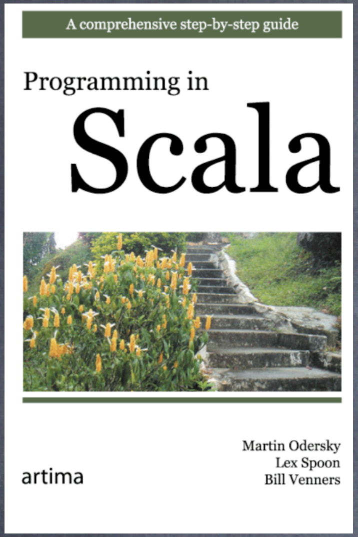
eSCALAtion of Usage



Seamless integration

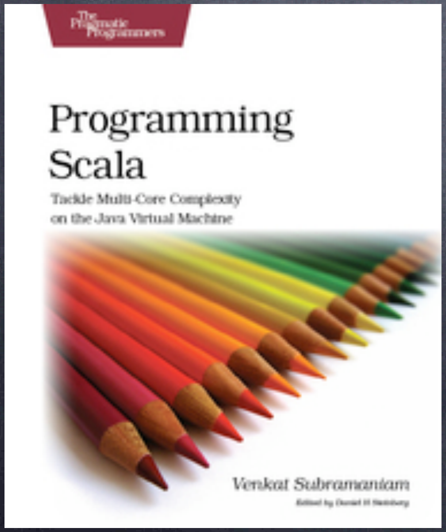
- Can call into any Java code
- Can call from any JVM language

References



http://booksites.artima.com/programming_in_scala

http://www.scala-lang.org


The Pragmatic Programmers
Programming
Scala
Tackle Multi-Core Complexity
on the Java Virtual Machine
Venkat Subramaniam

http://www.pragprog.com/titles/vsscala

Thank You!