Scala

Language Tour

```
m.map { t =>    val (s, i) = t; (s, i+1) }
```

```
m.map { t =>    val (s, i) = t; (s, i+1) }
```

<insert mental “record scratch” here>

Craig Tataryn

Basement CODERS

@craiger

GRIND Software Inc.

Craig Tataryn

- The Basement Coders Podcast
  - Basement CODERS - basementcoders.com

@craiger

GRIND Software Inc.

Craig Tataryn

@craiger

- The Basement Coders Podcast
  - - basementcoders.com
- User Groups
  - WfPG - wfpg.ca - wjpg.ca

GRIND Software Inc.

Craig Tataryn

@craiger

- The Basement Coders Podcast
  - - basementcoders.com
- User Groups
  - WfPG - wfpg.ca - wjpg.ca
- GRIND Software Inc.
  - grindsoftware.com

What is Scala?

What is Scala?

- Combines Functional & Object Oriented

What is Scala?

- Combines Functional & Object Oriented
- Interop with Java

What is Scala?

- Combines Functional & Object Oriented
- Interop with Java
- Dynamic-like Syntax in a Static Language

What is Scala?

- Combines Functional & Object Oriented
- Interop with Java
- Dynamic-like Syntax in a Static Language
- Easy to start playing with (REPL)

Let's Start

Variables & Classes

```
m.map { t =>    val (s, i) = t; (s, i+1) }
```

```
m.map { t =>    val (s, i) = t; (s, i+1) }
```

<insert mental “record scratch” here>

Declaring a Class

```
public class Person {
    public String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
      return this.name;
    }

    public void setName(String name) {
      this.name = name;
    }
}
```

```
class Person(var name:String)
```

*

Function Definitions

```
def savePerson(p: Person): ID = {
  dao.save(p)
}
```

Function Definitions

Start of Function Declaration

```
def savePerson(p: Person): ID = {
   dao.save(p)
}
```

Function Definitions

Start of Function Declaration

Function Name

```
def savePerson(p: Person): ID = {
  dao.save(p)
}
```

Function Definitions

Start of Function Declaration

Function Name

Parameter Name

```
def savePerson(p: Person): ID = {
  dao.save(p)
}
```

Function Definitions

Start of Function Declaration

Function Name

Parameter Name

Parameter Type

```
def savePerson(p: Person): ID = {
  dao.save(p)
}
```

Function Definitions

Start of Function Declaration

Function Name

Parameter Name

Parameter Type

Function Return Type

```
def savePerson(p: Person): ID = {
  dao.save(p)
}
```

Function Definitions

Start of Function Declaration

Function Name

Parameter Name

Parameter Type

Function Return Type

```
def savePerson(p: Person): ID = {
  dao.save(p)
}
```

Start of Function Definition

Function Definitions

Start of Function Declaration

Function Name

Parameter Name

Parameter Type

Function Return Type

```
def savePerson(p: Person): ID = {
  dao.save(p)
}
```

Start of Function Definition

Things to note:

- Last expression is return value
  - if more than one?
- = sign, the ; of Scala!
- : is used just like in variable declaration

Function Definitions

Start of Function Declaration

Function Name

Parameter Name

Parameter Type

Function Return Type

```
def savePerson(p: Person): ID = {
  dao.save(p)
}
```

Start of Function Definition

Things to note:

- Last expression is return value
  - if more than one?
- = sign, the ; of Scala!
- : is used just like in variable declaration

Function Definitions

Start of Function Declaration

Function Name

Parameter Name

Parameter Type

Function Return Type

```
def savePerson(p: Person): ID = {
  dao.save(p)
}
```

Start of Function Definition

Things to note:

- Last expression is return value
  - if more than one?
- = sign, the ; of Scala!
- : is used just like in variable declaration

Function Definitions

Start of Function Declaration

Function Name

Parameter Name

Parameter Type

Function Return Type

```
def savePerson(p: Person): ID = {
  dao.save(p)
}
```

Start of Function Definition

Things to note:

- Last expression is return value
  - if more than one?
- = sign, the ; of Scala!
- : is used just like in variable declaration

v.s. Java

v.s. Java

- Same file, multiple classes

v.s. Java

- Same file, multiple classes
- Constructor Chaining

v.s. Java

- Same file, multiple classes
- Constructor Chaining
- Static members

Objects

Objects vs Classes

*

Objects vs Classes

- We tend to think of

*

Objects vs Classes

- We tend to think of
  - Classes as templates

*

Objects vs Classes

- We tend to think of
  - Classes as templates
  - Objects as instances

*

Objects vs Classes

- We tend to think of
  - Classes as templates
  - Objects as instances
- In Scala

*

Objects vs Classes

- We tend to think of
  - Classes as templates
  - Objects as instances
- In Scala
  - object is a static template

*

Objects vs Classes

- We tend to think of
  - Classes as templates
  - Objects as instances
- In Scala
  - object is a static template
  - class is an instance templates

*

Traits

Traits

*

Traits

- Like an interface in Java

*

Traits

- Like an interface in Java
- Like an abstract class in Java

*

Traits

- Like an interface in Java
- Like an abstract class in Java
- Ability to “Mixin” traits

*

Operators

*

Looping

Control Structures

Control Structures

- The if statement, nothing new here

Control Structures

- The if statement, nothing new here
- Looping

Control Structures

- The if statement, nothing new here
- Looping
- while/do-while

Control Structures

- The if statement, nothing new here
- Looping
  - while/do-while
  - for comprehensions

Control Structures

- The if statement, nothing new here
- Looping
  - while/do-while
  - for comprehensions
  - Higher-Order List functions (foreach)

For Comprehensions

*

For Comprehensions

```
List<Employee> fullTime = new ArrayList<Employee>();
for (emp : empList) {
   if (emp.status == Consts.FULL_TIME)
      fullTime.add(emp);
}
```

*

For Comprehensions

```
List<Employee> fullTime = new ArrayList<Employee>();
for (emp : empList) {
   if (emp.status == Consts.FULL_TIME)
      fullTime.add(emp);
}
```

```
val fullTime = new ListBuffer[Int]
for (emp <- empList) {
   if (emp.status == Consts.FULL_TIME)
      fullTime += emp
}
```

*

For Comprehensions

```
List<Employee> fullTime = new ArrayList<Employee>();
for (emp : empList) {
   if (emp.status == Consts.FULL_TIME)
      fullTime.add(emp);
}
```

```
val fullTime = new ListBuffer[Int]
for (emp <- empList) {
   if (emp.status == Consts.FULL_TIME)
      fullTime += emp
}
```

```
val fullTime = for ( emp <- empList
      if emp.status == Consts.FULL_TIME) yield emp
```

*

Functions

Functions

Functions

- Functions are “kind of a big deal”

Functions

- Functions are “kind of a big deal”
- Higher Order

Functions

- Functions are “kind of a big deal”
- Higher Order
- First-class Types

Functions

- Functions are “kind of a big deal”
- Higher Order
- First-class Types
- Heterogeneously Typed

Functions

- Functions are “kind of a big deal”
- Higher Order
- First-class Types
- Heterogeneously Typed
  - number & types of parameters

Functions

- Functions are “kind of a big deal”
- Higher Order
- First-class Types
- Heterogeneously Typed
  - number & types of parameters
  - type of return value

Function Types

Function Types

Best described by showing Function literals

```
val add_one: (Int) => Int = (x) => x + 1
```

Function Types

Best described by showing Function literals

```
val add_one: (Int) => Int = (x) => x + 1
```

`add_one` is:

Function Types

Best described by showing Function literals

```
val add_one: (Int) => Int = (x) => x + 1
```

add_one is:

- A function which

Function Types

Best described by showing Function literals

```
val add_one: (Int) => Int = (x) => x + 1
```

add_one is:

- A function which
  - Takes a single parameter of type Int

Function Types

Best described by showing Function literals

```
val add_one: (Int) => Int = (x) => x + 1
```

add_one is:

- A function which
  - Takes a single parameter of type Int
  - Returns a value of Type Int

Function Types

Best described by showing Function literals

```
val add_one: (Int) => Int = (x) => x + 1
```

add_one is:

- A function which
  - Takes a single parameter of type Int
  - Returns a value of Type Int
- It's implementation is as follows

Function Types

Best described by showing Function literals

```
val add_one: (Int) => Int = (x) => x + 1
```

add_one is:

- A function which
  - Takes a single parameter of type Int
  - Returns a value of Type Int
- It's implementation is as follows
  - The parameter will be named x

Function Types

Best described by showing Function literals

```
val add_one: (Int) => Int = (x) => x + 1
```

add_one is:

- A function which
  - Takes a single parameter of type Int
  - Returns a value of Type Int
- It's implementation is as follows
  - The parameter will be named x
  - It will return x + 1

Function Types

Best described by showing Function literals

```
val add_one: (Int) => Int = (x) => x + 1
```

add_one is:

- A function which
  - Takes a single parameter of type Int
  - Returns a value of Type Int
- It's implementation is as follows
  - The parameter will be named x
  - It will return `x + 1`

map

map

- Not Map

map

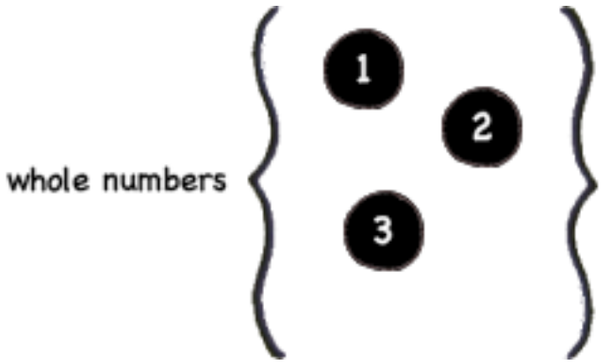
- Not Map
- A higher-order function

map

- Not Map
- A higher-order function
- Name is from Maths

"map an element from one set to another"

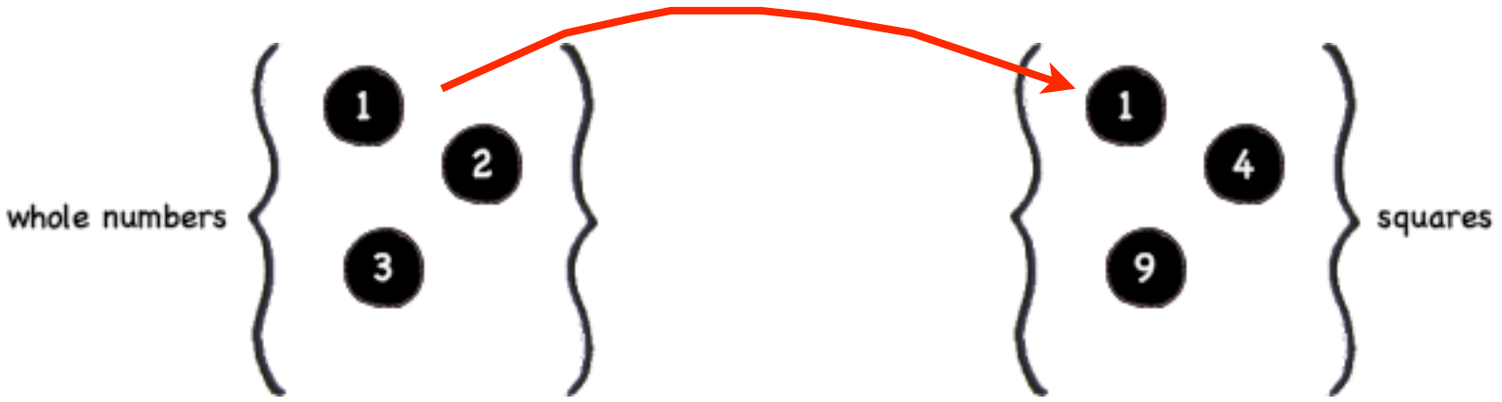
**


whole numbers
1
2
3

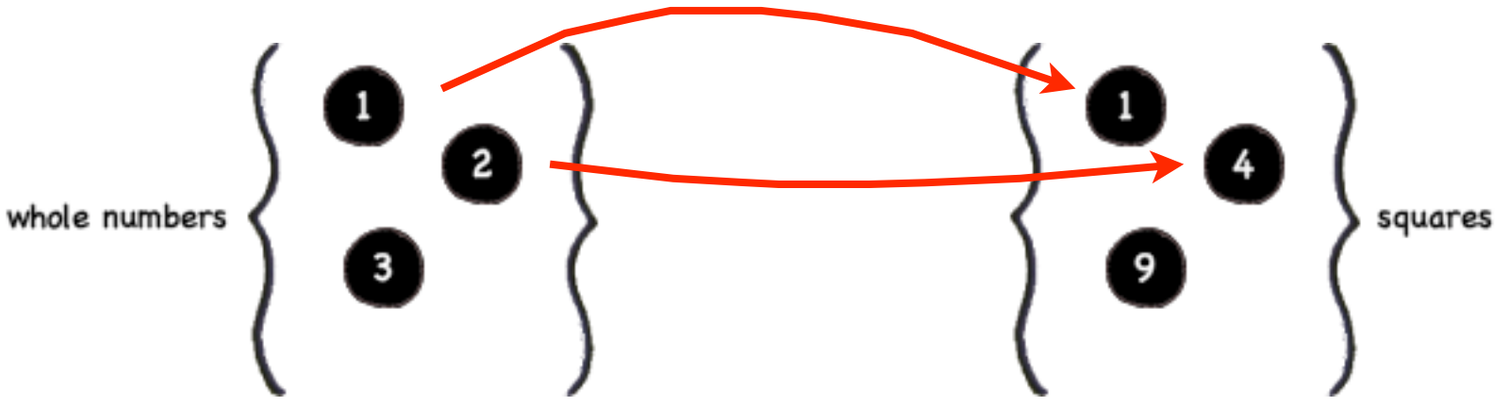
**


whole numbers
1
2
3
1
4
9
squares

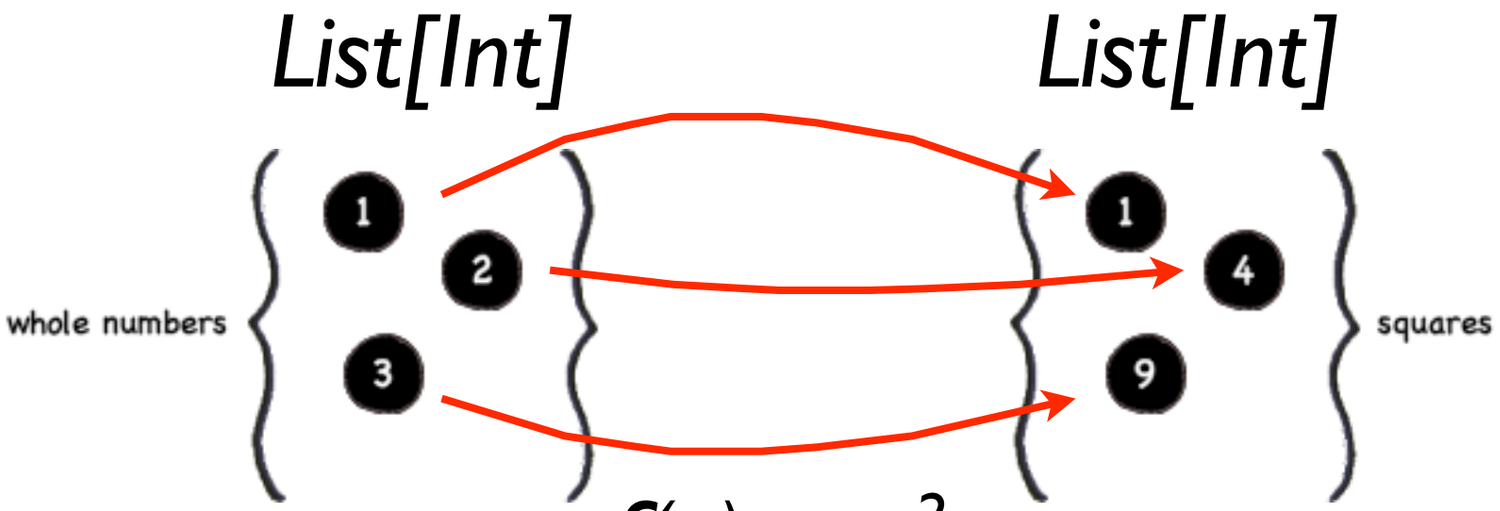
**


whole numbers
1
2
3
squares
1
4
9

**

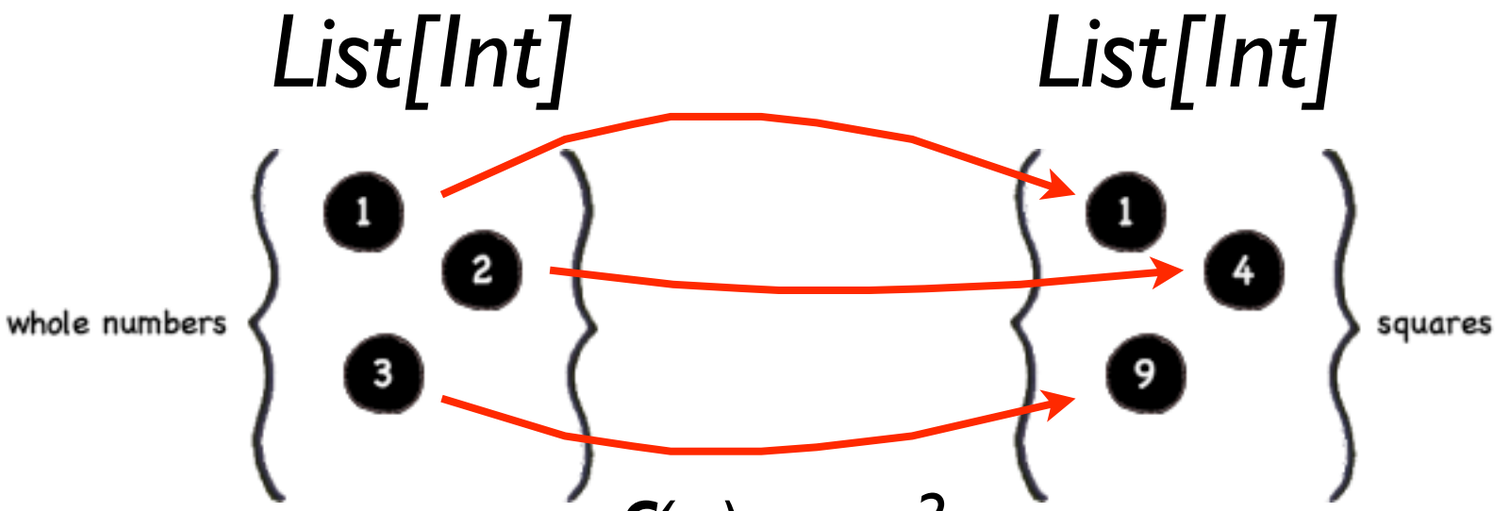

whole numbers
1
2
3
squares
1
4
9

**



$f(x) = x^2$

**



$f(x) = x^2$

```
val squares = wholeNumbers map(x => x * x)
```

**

FP Paradigm Shift

FP Paradigm Shift

- With Collections you begin to see the power of Higher-Order functions

FP Paradigm Shift

- With Collections you begin to see the power of Higher-Order functions
- You start viewing Data Structures as things you work with, not on.

FP Paradigm Shift

- With Collections you begin to see the power of Higher-Order functions
- You start viewing Data Structures as things you work with, not on.
  - You pass functions to Lists, rather than Lists to functions

Pattern Matching

case & match

*

case & match

- The case and match keywords are comprehenders just like for

*

case & match

- The case and match keywords are comprehenders just like for
- match comprehends *Partial Functions*

*

case & match

- The case and match keywords are comprehenders just like for
- match comprehends *Partial Functions*
- case statements generate *Partial Functions*

*

case & match

- The case and match keywords are comprehenders just like for
- match comprehends *Partial Functions*
- case statements generate *Partial Functions*
  - For several different types (*Strings, Ints, Regular Expressions, Types, Sequences, Foo...*)

*

Match on Foo

Match on Foo

In order to Allow:

```
var kid = Person("Mitch", "Tataryn")
 kid match {
     case Person("Mitch", "Tataryn") => println("Hi Son!")
     case Person("Lilja", "Tataryn") => println("Hi Daughter!")
     case Person(_,_) => println("Who are you?")
 }
```

Match on Foo

In order to Allow:

```
var kid = Person("Mitch", "Tataryn")
 kid match {
     case Person("Mitch", "Tataryn") => println("Hi Son!")
     case Person("Lilja", "Tataryn") => println("Hi Daughter!")
     case Person(_,_) => println("Who are you?")
 }
```

You can do:

```
object Person {
   def unapply(p: Person): (String, String) = {
     (p.fname, p.lname)
   }
}
```

Match on Foo

In order to Allow:

```
var kid = Person("Mitch", "Tataryn")
 kid match {
     case Person("Mitch", "Tataryn") => println("Hi Son!")
     case Person("Lilja", "Tataryn") => println("Hi Daughter!")
     case Person(_,_) => println("Who are you?")
 }
```

You can do:

```
object Person {
   def unapply(p: Person): (String, String) = {
     (p.fname, p.lname)
   }
}
```

Match on Foo

In order to Allow:

```
var kid = Person("Mitch", "Tataryn")
 kid match {
     case Person("Mitch", "Tataryn") => println("Hi Son!")
     case Person("Lilja", "Tataryn") => println("Hi Daughter!")
     case Person(_,_) => println("Who are you?")
 }
```

You can do:

```
object Person {
   def unapply(p: Person): (String, String) = {
     (p.fname, p.lname)
   }
}
```

Or:

```
case class Person(fname: String, lname: String)
```

Match on Foo

In order to Allow:

```
var kid = Person("Mitch", "Tataryn")
 kid match {
     case Person("Mitch", "Tataryn") => println("Hi Son!")
     case Person("Lilja", "Tataryn") => println("Hi Daughter!")
     case Person(_,_) => println("Who are you?")
 }
```

You can do:

```
object Person {
   def unapply(p: Person): (String, String) = {
     (p.fname, p.lname)
   }
}
```

Or:

```
case class Person(fname: String, lname: String)
```

The Tuple

Use a Tuple if...

*

Use a Tuple if...

- Want to return more than one thing

*

Use a Tuple if...

- Want to return more than one thing
- Want to extract more than one thing

*

About Tuples in Scala

```
m.map { t =>   val (s, i) = t; (s, i+1) }
```

About Tuples in Scala

- Tuples are Fundamental
- You'll start to recognize their use everywhere

```
m.map { t =>   val (s, i) = t; (s, i+1) }
```

If we have Time...

by Name

Parameters

*

- Gives you the ability to pass a code-block directly as a parameter

*

- Gives you the ability to pass a code-block directly as a parameter
- The code-block is "named" by the parameter inside the function

*

- Gives you the ability to pass a code-block directly as a parameter
- The code-block is "named" by the parameter inside the function

*

Partially Applied

Functions

- Allows you to “fill in a parameter” at a later time

- Allows you to “fill in a parameter” at a later time
- Useful for stating locally-scoped default parameters

- Allows you to “fill in a parameter” at a later time
- Useful for stating locally-scoped default parameters

```
def draw(widget: Widget, canv: Canvas) = ...
```

- Allows you to “fill in a parameter” at a later time
- Useful for stating locally-scoped default parameters

```
def draw(widget: Widget, canv: Canvas) = ...
.
.
.
val myDraw = draw(_, myWindow.canvas)
myDraw(myWidget)
```

Closures

Scope Doggy-bag

*

Scope Doggy-bag

- Functions are usually able to access:

*

Scope Doggy-bag

- Functions are usually able to access:
  - their parameters

*

Scope Doggy-bag

- Functions are usually able to access:
  - their parameters
  - their enclosing/super class

*

Scope Doggy-bag

- Functions are usually able to access:
  - their parameters
  - their enclosing/super class
- Closures add the ability to “take your scope with you”

*

Curried Functions

Curried Functions

- If you see a function call like this:
  `geoLocate(40.827873)(85.341797)`

Curried Functions

- If you see a function call like this:
  `geoLocate(40.827873)(85.341797)`

- It was defined like this:

```
def geoLocate(lat: Double)(lng: Double) = …
```

Curried Functions

- If you see a function call like this:
  `geoLocate(40.827873)(85.341797)`

- It was defined like this:
  `def geoLocate(lat: Double)(lng: Double) = …`

- YouCurry non-curried functions
  using `.curried`

Implicit Functions

Type Conversion

Type Conversion

- Implicit functions are typically used for Type Conversion

Type Conversion

- Implicit functions are typically used for Type Conversion
- Remember this example?

```
var str = "Hello"
str = 10  //Bzzzzt!
```

Type Conversion

- Implicit functions are typically used for Type Conversion
- Remember this example?

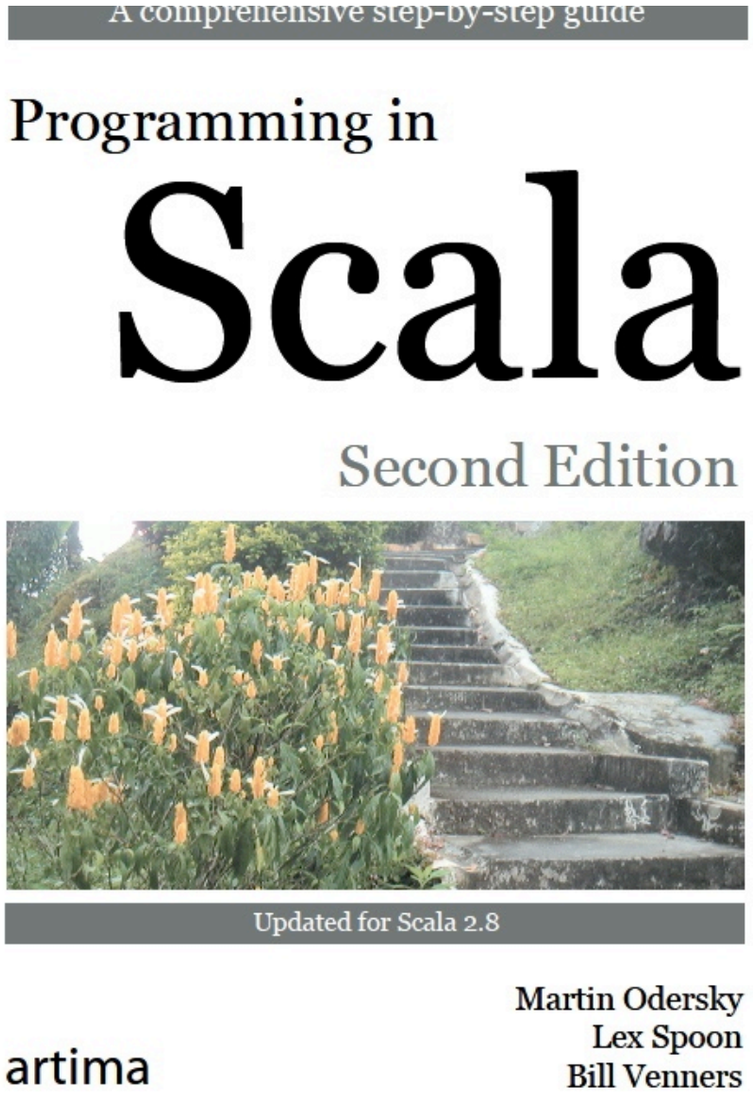
```
var str = "Hello"
str = 10  //Bzzzzt!

implicit def intToString(i: Int) {
  i.toString
}
```

References

Where to start

- Book
  - Programming **in** Scala (Artima)
  - Programming Scala (O'Reilly)
- StackOverFlow.com
- scala-user mailing list



Craig Tataryn


@craiger

craig@grindsoftware.com

git://github.com/ctataryn/ScalaLangTour.git

WfPG - wfpg.ca


- wjpg.ca

basementcoders.com