Introduction to Scala

SF-Scala Meetup
November 10th, 2016

Jason Swartz

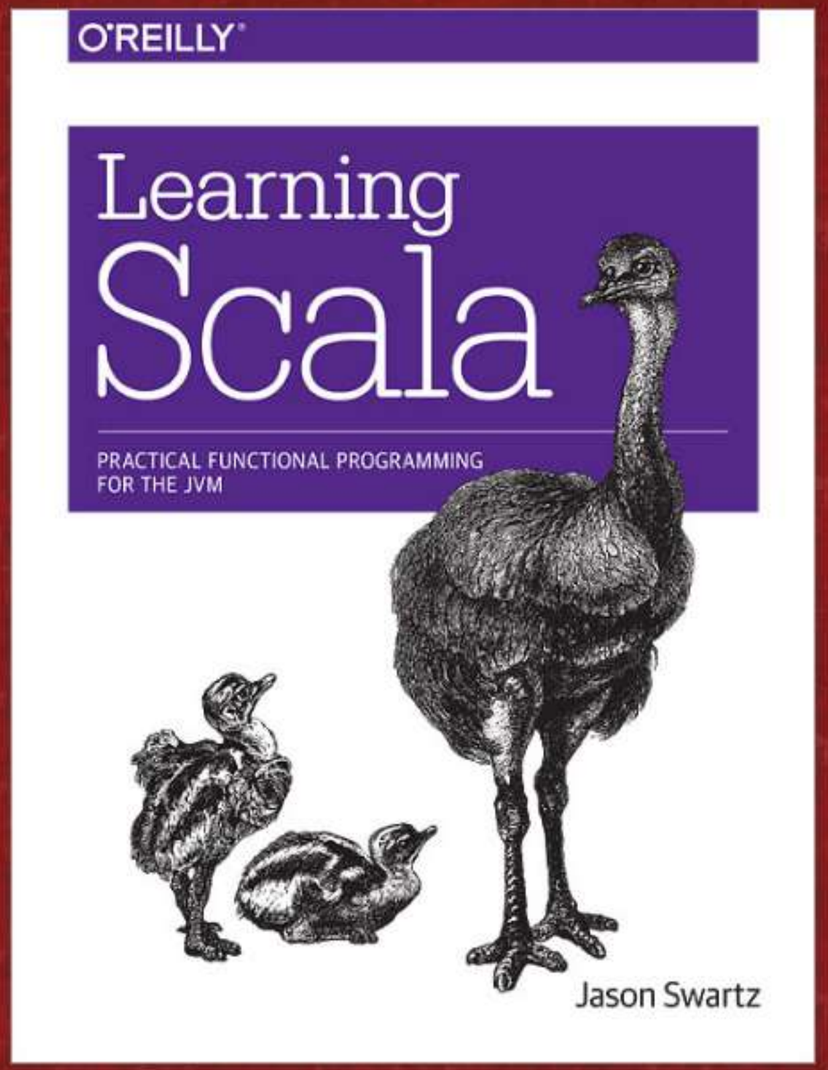
---

swartzrock at

NETFLIX

MESOSPHERE

CLASSPASS


O'REILLY
Learning
Scala
PRACTICAL FUNCTIONAL PROGRAMMING
FOR THE JVM
Jason Swartz

Please Install Scala

http://www.scala-lang.org/download

Lets get our environment ready

It's time to run

> scala

```
[6:46] zsh:jason@local ~  > scala
Welcome to Scala version 2.11.1 (Java HotSpot(TM)
64-Bit Server VM, Java 1.8.0_05).
Type in expressions to have them evaluated.
Type :help for more information.

scala> println("Hello, Scala")
Hello, Scala

scala>
```

```
println("Hello, Scala")
```

Part 1

Introduction

Now the learning begins

New Stuff

The Scala Language

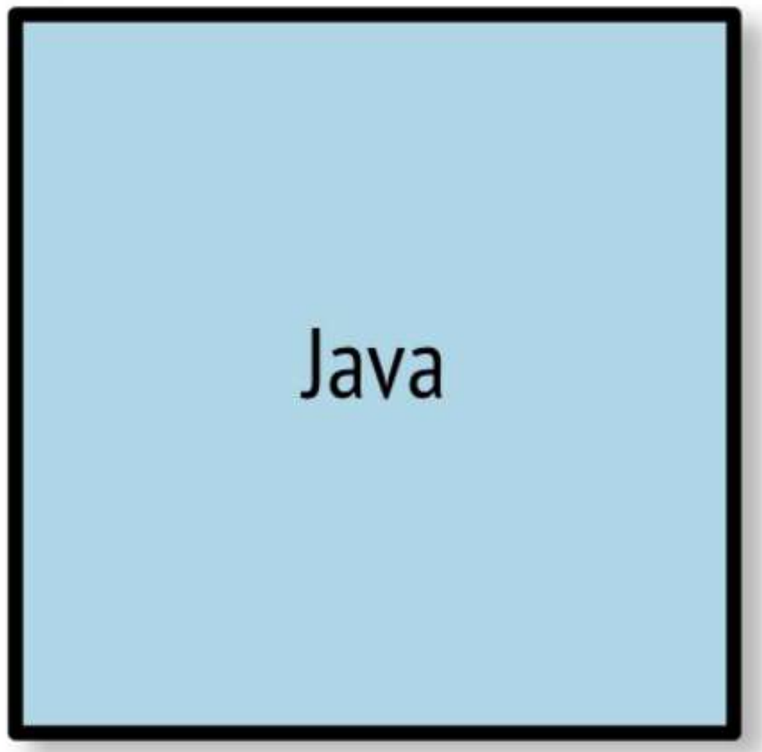
The Java Virtual Machine

First-Class Functions

Static Types & Generics

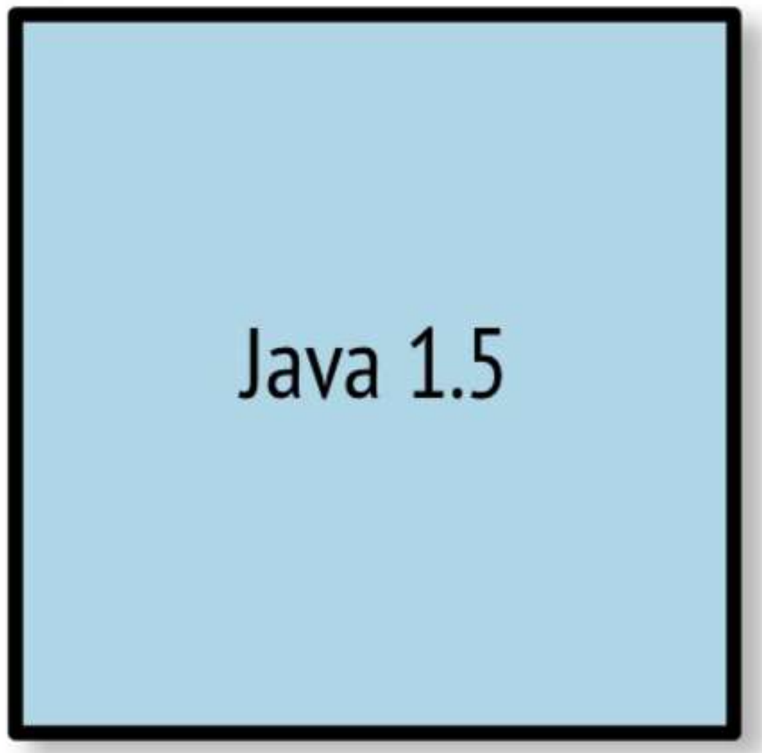
Immutable Data

Pattern Matching

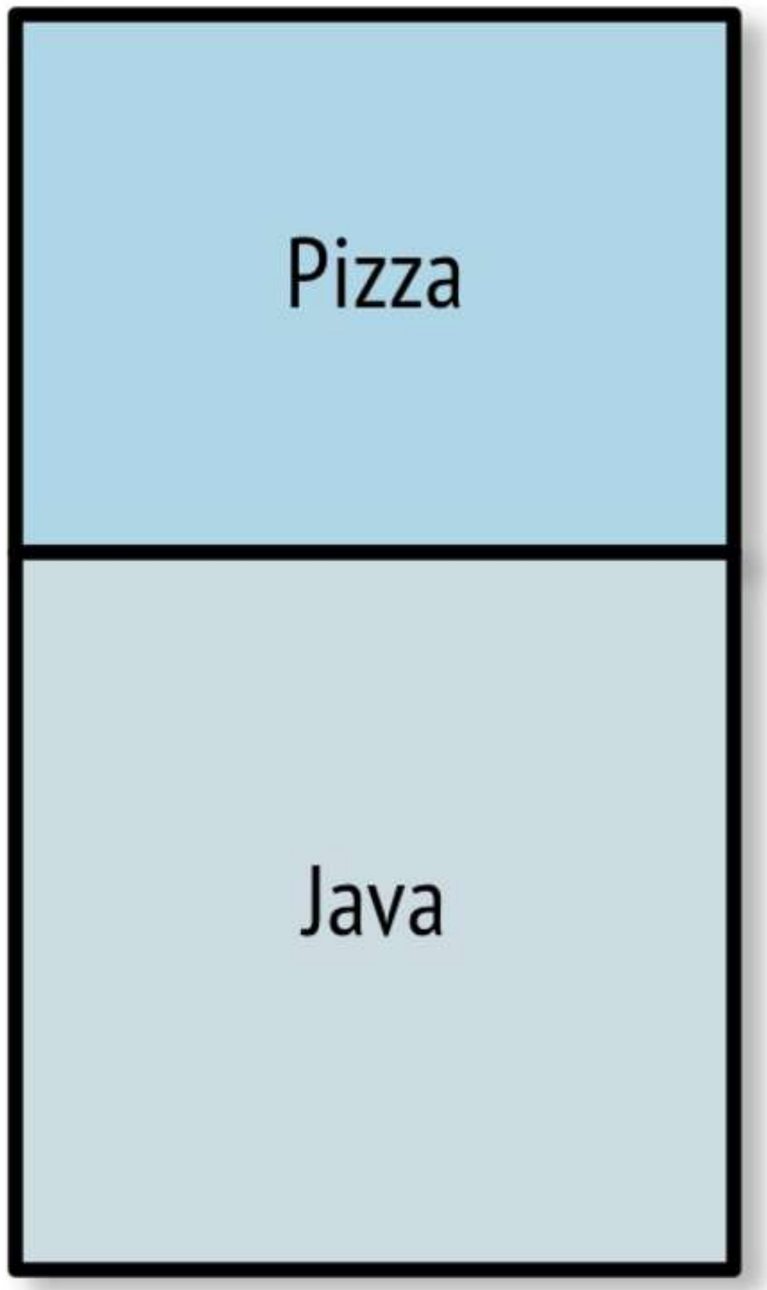


- Compiled
- Object-Oriented
- Static Types
- Fast (eventually)

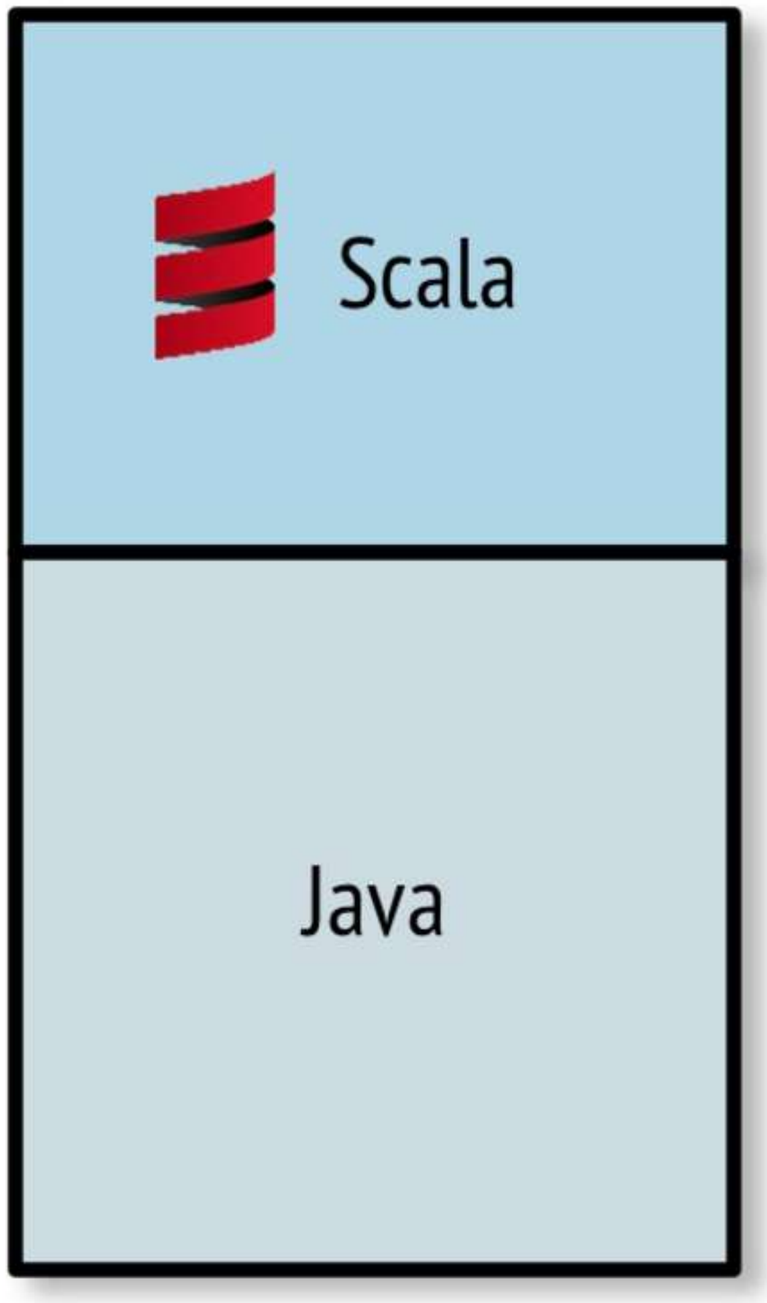
We Have So Much To Cover Today



- Generic Types
- Optimizing Compiler
- ~~First-Class Functions~~
- ~~Pattern Matching~~



- Generic Types
- Optimizing Compiler
- First-Class Functions
- Pattern Matching



- First-Class Functions
- Pattern Matching
- Immutable Data
- Awesome New Language

Java

```
String name = "Phil";
```

Scala

```
var name: String = "Phil";
```

Java

```
String name = "Phil";
```

Java

```
String name = "Phil";
```

Scala

```
var name           = "Phil"
```

Java

```
String name = "Phil";
```

Scala

```
var name: String = "Phil"
```

Java

```java
final String name = "Phil";
```

Java

```java
final String name = "Phil";
```

Scala

```scala
val name          = "Phil"
```

Java

```java
final String name = "Phil";
```

Scala

```scala
val name: String = "Phil";
```

Java

```java
int doubler(int amount) {
  return amount * 2;
}
```

Scala

```scala
def doubler(amount: Int): Int = {
  return amount * 2
}
```

Java

```
int doubler(int amount) {
   return amount * 2;
}
```

Java

```
int doubler(int amount) {
  return amount * 2;
}
```

Scala

```
def doubler(amount: Int): Int = amount * 2
```

Java

```
int doubler(int amount) {
  return amount * 2;
}
```

Scala

```
def doubler(amount: Int): Int = {
  amount * 2
}
```

Java

```java
int doubler(int amount) {
  return amount * 2;
}
```

Scala

```scala
def doubler(amount: Int) = amount * 2
```

Scala

```scala
case class Node(name: String)
```

Java

```
public class Node {
   public String name;
   public Node(String name) { this.name = name; }
   override public void String toString() {
      return name.toString();
   }
   override public boolean equals(Node n) {
      return name.equals(n.name);
}
```

Scala is

Mysterious

How would you describe Scala?

Scala is a JVM language

Scala is

~~Mysterious~~

Scala is Object-Oriented

Scala is Type Safe

Scala is Functional Programming

(whatever *that* means)

Scala is

Expression-Oriented

Scala is Expressive

Scala is built for

Speed

Scala is Limber

Scala is built for

Productivity

Scala is built for

Performance

Now let's learn some

Scala

Okay, that's enough about Scala

Time to REPL with Scala

Part 2

The Basics

Are we ready to

REPL ?

```
println("Hello, Scala")
```

scala -Dscala.color

```
"Hello, Scala"
```

```
[6:46] zsh:jason@local ~  > scala
Welcome to Scala version 2.11.1 (Java HotSpot(TM)
64-Bit Server VM, Java 1.8.0_05).
Type in expressions to have them evaluated.
Type :help for more information.

scala> println("Hello, Scala")
Hello, Scala

scala>
```

```
scala> println("Hello, Scala")
Hello, Scala

scala> "Hello, Scala"
res1: String = Hello, Scala
```

A Handy Value Containing The Result For Later Reuse

The Type Of The Result

Your Result, In Case You Forgot

```
scala> println("Hello, Scala")
Hello, Scala

scala> "Hello, Scala"
res1: String = Hello, Scala
```

Multiline Strings

```
"""
    Hello,

     World
"""

"""
html.replaceAll("""</?[^>]+>""", "")
```

Strings

```
"Hello, " + "World"

"Hello, World" == "Hello, World"

"Hello, World".size

"Hello, World".replaceAll("W.*", "Java")
```

Lists

```
List(1, 3, 20): List[Int]

List('a', 'b', 'c')

List("hi", "there")

List(x: A, y: A, z: A)
```

More Types

```
12: Int

3.1416: Double

true: Boolean

List(1, 3, 20): List[Int]
```

Lists

```
List(1, 3, 20): List[Int]

List('a', 'b', 'c'): List[Char]

List("hi", "there"): List[String]

List(x: A, y: A, z: A)
```

Lists

```
List(1, 3, 20): List[Int]

List('a', 'b', 'c'): List[Char]

List("hi", "there")

List(x: A, y: A, z: A)
```

Generics

defn: Code or data that takes a type as a parameter specifying a type to use at runtime

List[A] could be List[Int] or List[Char] or List[String]

Lists

```
List(1, 3, 20): List[Int]

List('a', 'b', 'c'): List[Char]

List("hi", "there"): List[String]

List(x: A, y: A, z: A): List[A]
```

Generics

defn: Code or data that takes a type as a parameter specifying a type to use at runtime

List[A] could be List[Int] or List[Char] or List[String] or List[List[Int]]

List Operations

```
List(1, 2) :+ 3

List(1, 2) ++ List(3, 4)
```

List Operations

```
List(1, 2) :+ 3
```

Scala Types

Name	Description	Example
Any	Base type	`n/a`
Boolean	Either true or false	`false`
Char	Unicode character	`'A'`
Double	8-byte floating point	`9.23`
Int	4-byte integer	`10`
List	Sequence of items	`List(1,2,3)`
Long	8-byte integer	`327L`
String	Text	`"Hello"`

List Operations

```
List(1, 2) :+ 3

List(1, 2) ++ List(3, 4)

List(2, 4, 1, 3).sorted
```

Syntax: Defining A Value

```
val identifier[: type] = expression
```

```
val count = 20 + 5
val eolChar = '|'
val names = List("Oona", "Lily")
```

Syntax: Defining A Value

```
val identifier[: type] = expression
```

That was Types & Values

Syntax: Defining A Value

```
val identifier[: type] = expression
```

```
val count: Int = 20 + 5
val eolChar: Char = '|'
val names: List[String] = List("Oona", "Lily")
```

Exercises

1. The formula for converting Centigrade to Fahrenheit is
   `f = c * 9/5 + 32`
   In the Scala REPL, convert 22.5° Centigrade to the equivalent temperature in Fahrenheit
2. Take the Fahrenheit result from the exercise above, halve it, and convert it back to Centigrade.
3. Create a file named "HelloWorld.scala" with a single println. Run it from the Scala REPL with this command:
   `:load HelloWorld.scala`

How about we try out your new skills?

```
scala> println("Hello, Scala")
Hello, Scala

scala> "Hello, Scala"
res1: String = Hello, Scala
```

Part 3

Expressions

Expressions are valid commands

How can **that** be valid code?

Expression

defn: A unit of code that returns a value after it has been executed.

"Hello, Scala"

Expression

defn: A unit of code that returns a value after it has been executed.

Expression

defn: A unit of code that returns a value after it has been executed.

"Hello, Scala" true 23 + 42 / 2

Expression

defn: A unit of code that returns a value after it has been executed.

"Hello, Scala" true

```
val x = 25
```

```
x * 2
```

Expression

defn: A unit of code that returns a value after it has been executed.

"Hello, Scala" true 23 + 42 / 2

loadUsersFromDatabase()

Expression Blocks

```
{
   val x = 25
   x * 2
}
```

Expression Blocks

```
val y = {
   val x = 25
   x * 2
}
```

```
val greeting = { "Hello" }
```

Expression Blocks

```
val y = {
  val x = 25
  x * 2
}
```

Expression Block

defn: An expression composed of one or more expressions surrounded by curley braces.

Expression Blocks

```
val y = {
   val x = 25
   x * 2
}
```

```
val greeting = { "Hello" }
```

```
val sum = { { { 2 } } }
```

Syntax: If-Else expressions

```
if (boolean) expression else expression
```

More Expressions

Syntax: If-Else expressions

```
if (boolean) expression else expression
```

```
if (x > 10) {
  println("more")
} else {
  println("less")
}
val msg = if (x > 10) { "more" } else { "less" }
```

Syntax: If-Else expressions

```
if (boolean) expression else expression
```

```
if (x > 10) {
  println("more")
} else {
  println("less")
}
```

Syntax: If-Else expressions

```
if (x < 0) {
  "less than"
}
else if (x > 0) {
  "greater than"
}
else {
  "about the same"
}
```

Syntax: If-Else expressions

```
if (boolean) expression else expression
```

```
if (x > 10) {
  println("more")
} else {
  println("less")
}
val msg = if (x > 10) { "more" } else { "less" }
val msg = if (x > 10) "more" else "less"
```

Expressions are Recursive

Syntax: If-Else expressions

```
if (x < 0) {
  "less than"
}
else
  if (x > 0) {
    "greater than"
  }
  else {
    "about the same"
  }
```

Syntax: If-Else expressions

```
if (boolean) expression
else expression
```

```
val sum = { { { 2 } } }
```

Syntax: If-Else expressions

```
if (x < 0) {
  "less than"
}
else if (x > 0) {
  "greater than"
}
else {
  "about the same"
}
```

Syntax: If-Else expressions

```
if (boolean) expression-block
else expression-block
```

Syntax: Match expressions

```
expression match {
  case pattern => expression
}
```

Expressive Pattern Matching

Syntax: Match expressions

```
expression match {
  case pattern => expression
}
```

```
val valid = parts match {
  case List(12, 24) => true
  case List(18, 32) => true
  case List(22, 28) => false
}
```

Syntax: Match expressions

```
expression match {
  case pattern => expression
}
```

```
val msg = status match {
  case 200 => "okay"
  case 400 => "not okay"
}
```

Syntax: Match expressions

```
expression match {
  case pattern => expression
}
```

```
val hex = color match {
  case "red"   => "#f00"
  case "green" => "#0f0"
  case "blue"  => "#0ff"
}
```

Syntax: Match expressions

```
expression match {
  case pattern => expression
}
```

```
val msg = 500 match {
  case 200 => "okay"
  case 400 => "not okay"
}
```

```
scala.MatchError: 500
```

Syntax: Match expressions

```
expression match {
   case pattern => expression
}
```

```
val msg = 500 match {
   case 200 => "okay"
   case 400 => "not okay"
}
```

Syntax: Value Binding

```
expression match {
   case pattern => expression
   case value   => expression
}
```

Pattern Matching with Value Binding

Syntax: Value Binding

```
expression match {
   case pattern => expression
   case value   => expression
}
```

```
val valid = parts match {
   case List(12, x ) => true
   case List(18, 32) => true
   case List(13, 28) => false
}
```

Syntax: Value Binding

```
expression match {
   case pattern => expression
   case value   => expression
}
```

```
val msg = 500 match {
   case 200 => "okay"
   case 400 => "not okay"
   case x   => "hmm got this: " + x
}
```

```
expression match {
   case pattern                 => expression
   case value                   => expression
}
```

Pattern Matching with Pattern Guards

Syntax: Pattern Guards

```
expression match {
   case value if boolean => expression
}
```

```
expression match {
  case pattern                => expression
  case value if boolean => expression
  case value                  => expression
}
```

Congrats on mastering

Expressions

How about we try out your new skills?

Syntax: Pattern Guards

```
expression match {
  case value if boolean => expression
}
```

```
val msg = status match {
  case 200 => "okay"
  case x if x < 500 => "odd, got this: " + x
  case x => "error! got this error status: " + x
}
```

Part 4

Functions

Exercises

1. Write a **match** expression that takes a string and returns it if the string is non-empty, or else returns "n/a".

2. Given a double value, write an **if-else** expression to return "greater" if it is more than zero, "same" if it equals zero, and "less" if it is less than zero. Can you write this as a **match** expression

Function

defn: An expression with a name & input parameters

What is a Function anyways?

```
{
  val now = System.currentTimeMillis
  val seconds = now / 1000
  "Num seconds since 1970 = " + seconds
}
```

```
val now = System.currentTimeMillis
val seconds = now / 1000
"Num seconds since 1970 = " + seconds
```

```
def numSeconds: String = {
  val now = System.currentTimeMillis
  val seconds = now / 1000
  "Num seconds since 1970 = " + seconds
}

val since1970: String = numSeconds()
```

```
val since1970: String = {
  val now = System.currentTimeMillis
  val seconds = now / 1000
  "Num seconds since 1970 = " + seconds
}
```

```
def numSeconds: String = {
  val now = System.currentTimeMillis
  val seconds = now / 1000
  "Num seconds since 1970 = " + seconds
}
```

```scala
def numSeconds(then: Long): String = {
  val now = System.currentTimeMillis
  val seconds = (now - then) / 1000
  "Num seconds since then = " + seconds
}

val sinceMay2014 = numSeconds(1400000000000L)
```

```scala
def numSeconds(then: Long): String = {
  val now = System.currentTimeMillis
  val seconds = (now - then) / 1000
  "Num seconds since then = " + seconds
}
```

Functions Are

Expressions

with names

Functions Are

Expressions

with names

and inputs

Functions Are

Expressions

Syntax: Functions

```
def name = expression
```

```
def greeting = "Hello, World"
val message = greeting + ", here I am!"

println(message)
```

Syntax: Functions

```
def name = expression
```

Syntax: Functions

```
def name[: type] = expression
```

```
def greeting: String = "Hello, World"
def message: String = greeting + ", here I am!"

println(message)
```

Syntax: Functions

```
def name(name1: type1[, ]) = expression
```

```
def divide(a: Int, b: Int) = {
  if (b == 0) 0
  else a / b
}

val speed = divide(miles, hours)
```

Syntax: Functions

```
def name = expression
```

```
def greeting = "Hello, World"
def message = greeting + ", here I am!"

println(message)
```

Now You Know

Functions

Syntax: Functions

```
def name(name1: type1[, ]): type = express
```

```
def divide(a: Int, b: Int): Int = {
  if (b == 0) 0
  else a / b
}

val speed: Int = divide(miles, hours)
```

Exercises

1. The formula for computing the area of a circle given its radius is
   `area = math.Pi * (square of the radius)`
   Write a function that returns the area of a circle given its radius.

2. Write a function that takes a milliseconds value and returns a string describing the duration in days, hours, minutes and seconds. What's the optimal type for the input value?

Part 5

First-Class

Functions

How about we try out your new skills?

Functions are *also* Data

Functions are

Expressions

```
val myFunction
myFunction()

val myCopy = myFunction
myCopy()
```

```
val myFunction
myFunction()

val myCopy = myFunction
myCopy()

def invoke(someFunction) = someFunction()
invoke( myCopy )
```

```
val myFunction
myFunction()
```

Functions as data are Storable

Functions as data are

Passable

What does this imply?

If Scala data has a Type...

Functions as data are Typed

```
def buzz(x: Int): String = {
  "the value is " + x
}
```

```scala
def buzz(x: Int): String = {
  "the value is " + x
}
```

```scala
val fizz: (Int) => String = buzz
```

What is the type of a function?

Syntax: Function Types

```
(type[, type, ...]) => output-type
```

```scala
def buzz(x: Int): String = {
  "the value is " + x
}
```

```scala
val fizz: (Int) => String = buzz

fizz(3)
```

Leave The Function At

Home

Syntax: Function Literals

```
(id: type) => expression
```

Syntax: Function Types

```
(type[, type, ...]) => output-type
```

```
def hello(name: String) = "Hello, " + name

val h: String => String = hello
h("World")
```

Syntax: Function Literals

```
(id: type) => expression
```

```
val l = (name: String) => {
    val nameLength = name.length
    "Your name is " + nameLength + " letters long!"
}
```

Syntax: Placeholder Syntax

```
val doubler = (x: Int) => x * 2
val doubled = doubler(22)
```

Syntax: Function Literals

```
(id: type) => expression
```

```
val doubler = (x: Int) => x * 2
val doubled = doubler(22)
```

Syntax: Placeholder Syntax

```
val h: String => String = "Hello, " + _
h("World")
```

Syntax: Placeholder Syntax

```
val doubler = (x: Int) => x * 2
val doubled = doubler(22)
```

```
val doubler = (x: Int) => _ * 2
val doubled = doubler(22)
```

Remember, functions are

Passable

Higher-Order Functions

defn: A function that takes another function as input

Syntax: Placeholder Syntax

```scala
val h: String => String = "Hello, " + _
h("World")
```

```scala
val f: (Int, Int) => Int = _ + _
f(4, 6)
```

```
def reverser(s: String) = s.reverse
```

```
val rev = reverser(userName)
```

```
def stringSafe(s: String, f: String => String) = {
  if (s != null) f(s) else s
}
```

```
def reverser(s: String) = s.reverse

val rev = stringSafe(userName, reverser)
```

Higher-Order Functions

defn: A function that takes another function as input
or as a return value

```
def stringSafe(s: String, f: String => String) = {
  if (s != null) f(s) else s
}
```

```
// def reverser(s: String) = s.reverse

// val rev = stringSafe(userName, reverser)
val rev = stringSafe(userName, s => s.reverse)
```

Can we make this Simpler?

Now You Know

Higher-Order Functions

How about we try out your new skills?

```
def stringSafe(s: String, f: String => String) = {
  if (s != null) f(s) else s
}
```

```
// def reverser(s: String) = s.reverse

// val rev = stringSafe(userName, reverser)
val rev = stringSafe(userName, _.reverse)
```

Part 6

Collections

Collections

defn: Data structures for collecting values of a given type

Exercises

1. The format for function literals is:
   `(id: type) => expression`
   Write a function literal that takes two integers and returns the higher number.

2. Write a higher-order function that takes 4 parameters: 3 integers and a `(Int, Int) => Int` function argument. It should use the function argument with the integers to pick a single winner and return it.

   Try invoking the function with the function literal created in the first exercise.

Lists

Immutable linked lists

Immutable Collections

defn: Collections which cannot be modified after they are created.

List Operations

```
scala> val a = List(1, 2)
a: List[Int] = List(1, 2)

scala> val b = a :+ 3
b: List[Int] = List(1, 2, 3)

scala> a
res16: List[Int] = List(1, 2)
```

More List Operations

```
val rgb: List[String] = List("red", "green", "blue"
rgb(1)    // "red"
rgb.head // "red"
rgb.tail // List("green", "blue")
```

List Operations

```
List(1, 2) :+ 3

List(1, 2) ++ List(3, 4)

List(2, 4, 1, 3).sorted
```

More List Operations

```
val rgb: List[String] = List("red", "green", "blue"

val pgb = "purple" :: rgb.tail
```

More List Operations

```
val rgb: List[String] = List("red", "green", "blue"
rgb(1)    // "red"
rgb.head // "red"
rgb.tail // List("green", "blue")

rgb.tail.head // "green"
rgb.tail.tail // "blue"
```

Lists

Recursive immutable linked lists

Lists of a Higher Order

More List Operations

```
val rgb: List[String] = List("red", "green", "blue"
```

```
val pgb = "purple" :: rgb.tail
```

```
scala> println(rgb)
List(red, green, blue)
```

map()

```
val colors = List("red", "green", "blue")

colors.map( (c: String) => c.size )

colors.map( _.size )
```

reduce()

```
val colors = List("red", "green", "blue")
```

```
colors.reduce((a,i) => a + ", " + i)
```

map()

```
val colors = List("red", "green", "blue")

colors.map( (c: String) => c.size )
```

foreach()

```
val colors = List("red", "green", "blue")
colors.foreach( c => println("Roses are " + c) )
```

reduce()

```
val nums = List(52.3, 273.1, 92.9, 22.8)
```

```
nums.reduce((a,i) => if (a > i) a else i)
```

Exercises

1. You can use this command to create a list of the first 100 integers:`(1 to 100).toList`
   Can you convert this into a list of the first 20 odd numbers?

2. Write a function that takes a list of strings and returns the longest string in the list

Part 7

Classes

```
val colors = List("red", "green", "blue")
```

```
val shorties = x colors.filter(_.size < 5)
```

Scala is Object-Oriented

Class

defn: A bunch of data plus functions which act on that data

Now it's time we got

Classy

Syntax: Classes

```
class name {
  expressions,
  values &
  methods
}
```

Class

defn: A bunch of data plus functions which act on that data

ok, *methods* which act on that data

```scala
class User {
  val name: String = "Yubaba"
  def greet = "Hello from " + name
}
```

```
class User {
  val name: String = "Yubaba"
  def greet = "Hello from " + name
}
```

```
val u = new User()
println( u.greet )
println( "name = " + u.name )
```

Syntax: Classes

```
class name {
  expressions,
  values &
  methods
}

new name()
```

Class

defn: A bunch of data plus functions which act on that data

ok, *methods* which act on that data

ok, the data may be passed as *parameters*

Class

defn: A bunch of data plus functions which act on that data

ok, *methods* which act on that data

Syntax: Classes

```
class name(name: type[, ...]) {
  expressions,
  values &
  methods
}

new name(value)
```

```scala
class User(n: String) {
  val name: String = n
  def greet = "Hello from " + name
}
```

Syntax: Classes

```
class name(name: type[, ...]) {
  expressions,
  values &
  methods
}
```

Syntax: Classes

```
class name(val name: type[, ...]) {
  expressions,
  values &
  methods
}
```

Syntax: Classes

```
class name(val name: type[, ...]) {
  expressions,
  values &
  methods
}

new name(value)
```

```
class User(n: String) {
  val name: String = n
  def greet = "Hello from " + name
}
```

```
val u = new User("Zeniba")
println( u.greet )
println( "name = " + u.name )
```

```
class User(val name: String) {
  def greet = "Hello from " + name
}
```

```
val u = new User("Zeniba")
println( u.greet )
println( "name = " + u.name )
```

```
class User(val name: String) {
  def greet = "Hello from " + name
}
```

Syntax: Extending Classes

```
class Child extends Parent {}
```

```
class Child extends Parent {
  def method {}

  override def method { super.method() }
}
```

Syntax: Extending Classes

```
class A {
  def hi = "Hi from A"
}

new A().hi // Hi from A
```

Subtyping & Polymorphism

What's better than Classes?

Case Classes

Syntax: Extending Classes

```
class A {
  def hi = "Hi from A"
}

new A().hi // Hi from A
```

```
class B extends A {
  override def hi = { "B says " + super.hi }
}

new B().hi // B says Hi from A
```

Case Class

defn: A class with special features for holding data where class parameters are automatically fields

Case Class

defn: A class with special features for holding data

Case Class

defn: A class with special features for holding data
where class parameters are automatically fields
and is easily printable
and comparable

Syntax: Case Classes

```
case class Name()
```

Case Class

defn: A class with special features for holding data
where class parameters are automatically fields
and is easily printable

Syntax: Case Classes

```
case class Name()

case class User(name: String)
val u = User("Hadrian")

println(u) // User(Hadrian)
```

Syntax: Case Classes

```
case class Name()

case class User(name: String)
val u = User("Hadrian")
```

It's Time For

Singleton's


THE SINGLETON
AGED 12 YEARS
OF DUFFTOWN
TRADE MARK
THE SINGLETON
Single Malt Scotch Whisky

Syntax: Case Classes

```
case class Name()

case class User(name: String)
val u = User("Hadrian")

println(u) // User(Hadrian)

u == User("Royce") // false
```

Singleton

defn: A class that may have zero or one instances

Object

defn: A class that may have zero or one instances

Singleton

Syntax: Objects

```
object name

object HtmlUtils {
  def removeMarkup(input: String) = {
    input.replaceAll("""</?\w[^>]*>""","")
         .replaceAll("<.*>","")
  }
}
```

Syntax: Objects

```
object name

object HtmlUtils {
  def removeMarkup(input: String) = {
    input.replaceAll("""</?\w[^>]*>""","")
         .replaceAll("<.*>","")
  }
}

HtmlUtils.removeMarkup( htmlText )
```

Syntax: Objects

```
object name
```

and Case

Classes

We have covered

Classes

It's Time For

Singleton's

Wait! I have a much better idea!

plus Objects

Exercises

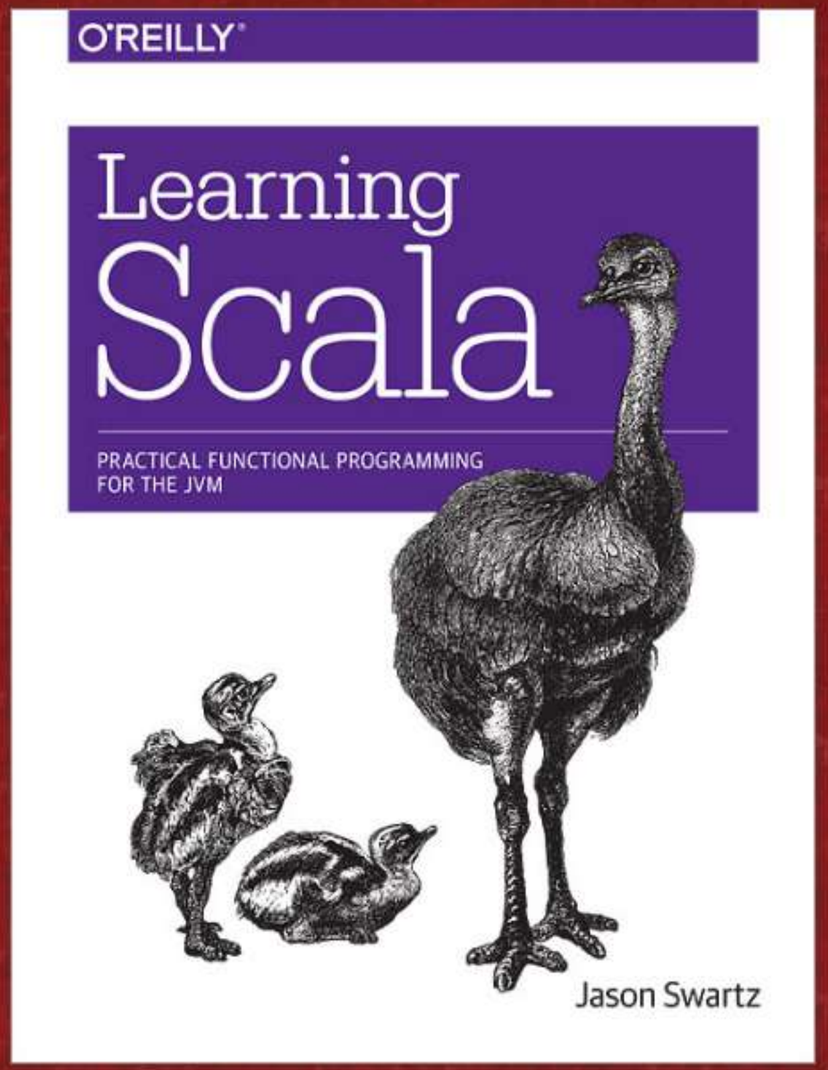
1. A case class can be created with the following syntax:
   `case class Thing(a: Int, b: String)`
   Create a video gaming console class that can track the make, model, debut date, and physical media formats supported. Can you compare two instances of this class?

2. Write a function that takes a list of instances of the video gaming console class and prints out the name of the most recent console.

Introduction to Scala

SF-Scala Meetup
November 10th, 2016

How about we try out your new skills?


O'REILLY
Learning
Scala
PRACTICAL FUNCTIONAL PROGRAMMING
FOR THE JVM
Jason Swartz

Jason Swartz

---

swartzrock at

NETFLIX

 MESOSPHERE

CLASSPASS

Thanks Everyone!

Q & A