The Scala Experience

Bill Venners
President
Artima, Inc.

Ted Neward
The Man
Neward and Associates
Overall Presentation Goal

Teach you enough Scala so that you can start using it for real work.
Enough about me (and Ted)

- Run the Artima Developer website
- Existing investment in Java knowledge and code
- Wanted a more productive language for JVM
- Didn't want to give up static typing
- Scala fit my needs
- Coauthored and published *Programming in Scala*
- Primary developer of ScalaTest
- And about Ted...
1. First Steps in Scala
$ scala
Welcome to Scala version 2.7.2.
Type in expressions to have them evaluated. Type :help for more information.
scala> 1 + 2
res0: Int = 3
scala> res0 * 3
res1: Int = 9
The classic:

```scala
scala> println("Hello, world!")
Hello, world!
```
Defining variables:

scala> val msg = "Hello, world!"
msg: java.lang.String = Hello, world!

scala> val msg: java.lang.String = "Hello, world!"
msg: java.lang.String = Hello, world!

scala> val msg: String = "Hello, world!"
msg: java.lang.String = Hello, world!
Vals and vars:

scala> println(msg)
Hello, world!

scala> msg = "Goodbye, cruel world!"
<console>:5: error: reassignment to val
   msg = "Goodbye cruel world!"

scala> var greeting = "Hello, world!"
greeting: java.lang.String = Hello, world!

scala> greeting = Leave me alone, world!"
greeting: java.lang.String = Leave me alone, world!
Odds and ends:

scala> val multiLine =
   |   "This is the next line."
multiLine: java.lang.String = This is the next line.

scala> val oops
   |   |
   |   |
   |   |
You typed two blank lines. Starting a new command.
Java's if statement:

```java
if (a > b)
    System.out.println(a);
else
    System.out.println(b);
```

Java's ternary operator:

```java
int m = (a > b) ? a : b;
System.out.println(m);
```
Scala’s if expression:

```scala
def printIn(m: Int): Unit = {
  println(m)
}
val m = if (a > b) a else b
println(m)
```
Defining a function:

```scala
scala> def max(x: Int, y: Int): Int = {
      |   if (x > y) x
      |   else y
      | }
      | max: (Int,Int)Int

scala> max(3, 5)
res6: Int = 5
```
Streamlining a function:

scala> def max2(x: Int, y: Int) = if (x > y) x else y
max2: (Int,Int)Int

scala> def greet() = println("Hello, world!")
greet: ()Unit

scala> greet()
Hello, world!

scala> :quit
$
In a file named helloarg.scala:

```
// Say hello to the first argument
println("Hello, " + args(0) + "!")
```

$ scala helloarg.scala planet
Hello, planet!
While loop:

```scala
var i = 0
while (i < args.length) {
    println(args(i))
    i += 1
}
```

$ scala printargs.scala Scala is fun Scala is fun
Another while loop:

```scala
var i = 0
while (i < args.length) {
    if (i != 0)
        print(" ")
    print(args(i))
    i += 1
}
println()
```

$ scala echoargs.scala
Scala is even more fun

Scala is even more fun
"Looping" with foreach:

```scala
args.foreach(arg => println(arg))

$ scala pa.scala Concise is nice
Concise
Is
nice

args.foreach((arg: String) => println(arg))

args.foreach(println)
```
"Looping" with a for expression:

```java
for (arg <- args) 
println(arg)
```
Exercises

1. Print hello world from the interpreter.
2. Print hello world from a script.
3. In the interpreter, define a function that takes a string and an Int, and prints the string the Int number of times.
4. Write a script that prints out its arguments on one line, but with each argument uppercased and reversed.
2. Next Steps in Scala
val greetStrings = new Array[String](3)

greetStrings(0) = "Hello"
greetStrings(1) = ", , "
greetStrings(2) = "world!\n"

for (i <- 0 to 2)
    print(greetStrings(i))
All operations are method calls:

\[
1 + 2 = (1) + (2)
\]
apply and update

greetStrings(i)
greetString.apply(i)

greetStrings(0) = "Hello"
greetStrings.update(0, "Hello")
Creating and initializing an array

```scala
val numNames = Array("zero", "one", "two")

val numNames2 = Array.apply("zero", "one", "two")
```
Creating and initializing a list

```scala
def main(_): Unit = {
  val oneTwoThree = List(1, 2, 3)
}
```
val oneTwo = List(1, 2)
val threeFour = List(3, 4)
val oneTwoThreeFour = oneTwo ::: threeFour
println(""+ oneTwo +" and "+ threeFour +" were not mutated.")
println("Thus, "+ oneTwoThreeFour +" is a new list.")

List(1, 2) and List(3, 4) were not mutated.
Thus, List(1, 2, 3, 4) is a new list.

Lists are immutable
Consing lists

```scala
val twoThree = List(2, 3)
val oneTwoThree = 1 :: twoThree
println(oneTwoThree)
```

List(1, 2, 3)

1 :: twoThree

twoThree.(1)
Initializing lists with cons and Nil

```scala
val oneTwoThree = 1 :: 2 :: 3 :: Nil
println(oneTwoThree)
List(1, 2, 3)
```
Creating and using a tuple

```
val pair = (99, "Luftballons")
println(pair._1)
println(pair._2)
```

99
Luftballons
Tuple types

(99, "Luftballons")
Tuple2[Int, String]

('u', 'r', "the", 1, 4, "me")
Tuple6[Char, Char, String, Int, Int, String]
Set hierarchy

scala.collection
Set
«trait»

scala.collection.immutable
Set
«trait»

scala.collection.mutable
Set
«trait»

scala.collection.immutable
HashSet

scala.collection.mutable
HashSet
Creating, initializing, and using an immutable set

```kotlin
var jetSet = Set("Boeing", "Airbus")
jetSet += "Lear"
println(jetSet.contains("Cessna"))
jetSet = jetSet + "Lear"
```
import scala.collection.mutable.Set

val movieSet = Set("Hitch", "Poltergeist")

movieSet += "Shrek"
println(movieSet)
If you need a set class other than the default

```scala
import scala.collection.immutable.HashSet

val hashSet = HashSet("Tomatoes", "Chilies")
println(hashSet + "Coriander")
```
import scala.collection._

val mut = mutable.Set(1, 2, 3)
val imm = immutable.Set(4, 5, 6)
import scala.collection.mutable.Map

val treasureMap = Map[Int, String]()
treasureMap += (1 -> "Go to island.")
treasureMap += (2 -> "Find big X on ground.")
treasureMap += (3 -> "Dig.")
println(treasureMap(2))

Find big X on ground.
Implicit conversions

```scala
implicit def any2ArrowAssoc[A](x: A): ArrowAssoc[A] = new ArrowAssoc(x)

3 -> "Dig."

(3).->("Dig.")

any2ArrowAssoc(3).->("Dig.")
```
Creating, initializing, and using an immutable map

```scala
val romanNumeral = Map(
  1 -> "I", 2 -> "II", 3 -> "III", 4 -> "IV", 5 -> "V"
)
println(romanNumeral(4))
```
def printArgs(args: Array[String]): Unit = {
  var i = 0
  while (i < args.length) {
    println(args(i))
    i += 1
  }
}
More functional...

def printArgs(args: Array[String]): Unit = {
    for (arg <- args)
        println(arg)
}

def printArgs(args: Array[String]): Unit = {
    args.foreach(printIn)
}
def formatArgs(args: Array[String]): String =
    args.mkString("
")

println(formatArgs(args))

val res = formatArgs(Array("zero", "one", "two"))
assert(res == "zero\none\ntwo")
A balanced attitude for Scala programmers

Prefer vals, immutable objects and methods without side effects. Reach for them first. Use vars, mutable objects, and methods with side effects when you have a specific need and justification for them.
import scala.io.Source

if (args.length > 0) {
  for (line <- Source.fromFile(args(0)).getLines)
    print(line.length + " " + line)
}

else
  Console.err.println("Please enter filename")
$ scala countchars1.scala countchars1.scala

23 import scala.io.Source
1
23  if (args.length > 0) {
1
50  for (line <- Source.fromFile(args(0)).getLines)
36   print(line.length +" "+ line)
2 }
5 else
47  Console.err.println("Please enter filename")
Exercises

1. Write a script that takes two arguments, a filename and a line length. The script should read the file and print out only those lines greater than the specified line length.

2. Enhance the script so that it first prints out the lines longer than the specified length, followed by the lines less than or equal to the specified length.

3. If you haven’t done so already, try and get rid of the vars.
3. Classes and Objects
class ChecksumAccumulator {

    private var sum = 0

    def add(b: Byte): Unit = {
        sum += b
    }

    def checksum(): Int = {
        return ~(sum & 0xFF) + 1
    }
}
class ChecksumAccumulator {

    private var sum = 0

    def add(b: Byte) {
        sum += b
    }

    def checksum(): Int = {
        return ~(sum & 0xFF) + 1
    }
}
A more concise class definition

// In file ChecksumAccumulator.scala

class ChecksumAccumulator {
    private var sum = 0
    def add(b: Byte) { sum += b }
    def checksum(): Int = ~(sum & 0xFF) + 1
}


// In file ChecksumAccumulator.scala

import scala.collection.mutable.Map

object ChecksumAccumulator {

  private val cache = Map[String, Int]()

  def calculate(s: String): Int =
    if (cache.contains(s))
      cache(s)
    else {
      val acc = new ChecksumAccumulator
      for (c <- s)
        acc.add(c.toByte)
      val cs = acc.checksum()
      cache += (s -> cs)
      cs
    }
}

A companion object
Using a singleton object

`ChecksumAccumulator.calculate("Every value is an object.")`
/ In file Summer.scala

import ChecksumAccumulator.calculate

object Summer {
  def main(args: Array[String]) {
    for (arg <- args) {
      println(arg + ": " + calculate(arg))
    }
  }
}
Compiling and running

$ scalac ChecksumAccumulator.scala Summer.scala

$ fsc ChecksumAccumulator.scala Summer.scala

$ scala Summer of love
of: -213
love: -182
The primary constructor

class Rational(n: Int, d: Int) {

    public class Rational { // This is Java
        private int n;
        private int d;
        public Rational(int n, int d) {
            this.n = n;
            this.d = d;
        }
    }
}

class Rational(n: Int, d: Int) {
    println("Created "+ n + "/" + d)
}

scala> new Rational(1, 2)
Created 1/2
res0: Rational = Rational@90110a
Overriding toString

class Rational(n: Int, d: Int) {
    override def toString = n +"/"+ d
}

scala> val x = new Rational(1, 2)
x: Rational = 1/3
class Rational(n: Int, d: Int) {
    require(d != 0)
    override def toString = n + "/" + d
}

Checking preconditions
Can only access class parameters on this instance

class Rational(n: Int, d: Int) { // This won't compile
  require(d != 0)
  override def toString = n +"/"+ d
  def add(that: Rational): Rational =
    new Rational(n * that.d + that.n * d, d * that.d)
}

<console>:11: error: value d is not a member of Rational
    new Rational(n * that.d + that.n * d, d * that.d)
    ^
<console>:11: error: value d is not a member of Rational
    new Rational(n * that.d + that.n * d, d * that.d)
    ^
Adding fields

class Rational(n: Int, d: Int) {
    require(d != 0)
    val numer: Int = n
    val denom: Int = d
    override def toString = numer + "/" + denom
    def add(that: Rational): Rational =
        new Rational(
            numer * that.denom + that.numer * denom,
            denom * that.denom
        )
}

```python
def max(that: Rational) =
    if (this.lessThan(that)) that else this
```

Self references
Auxiliary constructors

class Rational(n: Int, d: Int) {
  require(d != 0)
  val numer: Int = n
  val denom: Int = d
  def this(n: Int) = this(n, 1) // auxiliary constructor
  override def toString = numer +"/"+ denom
  def add(that: Rational): Rational =
    new Rational(
      numer * that.denom + that.numer * denom,
      denom * that.denom
    )
}
class Rational(n: Int, d: Int) {
    require(d != 0)
    val numer: Int = n
    val denom: Int = d
    def this(n: Int) = this(n, 1) // auxiliary constructor
    override def toString = numer +"/"+ denom
    def + (that: Rational): Rational =
        new Rational(
            numer * that.denom + that.numer * denom,
            denom * that.denom
        )
}
Defining operators

scala> val x = new Rational(1, 2)
x: Rational = 1/2

scala> val y = new Rational(2, 3)
y: Rational = 2/3

scala> x + y
res8: Rational = 7/6
Overloading methods

class Rational(n: Int, d: Int) {
    require(d != 0)
    val numer: Int = n
    val denom: Int = d

def this(n: Int) = this(n, 1) // auxiliary constructor

override def toString = numer +"/"+ denom

def + (that: Rational): Rational =
    new Rational( 
        numer * that.denom + that.numer * denom, 
        denom * that.denom
    )

def + (i: Int): Rational =
    new Rational(numer + i * denom, denom)
}
Using overloaded methods

scala> val x = new Rational(2, 3)
x: Rational = 2/3

scala> x * x
res13: Rational = 4/9

scala> x + 1
res14: Rational = 4/3
What about the other way around?

```scala
scala> 2 * r
<console>:7: error: overloaded method value * with alternatives (Double)Double <and> (Float)Float <and> (Long)Long <and> (Int)Int <and> (Char)Int <and> (Short)Int <and> (Byte)Int cannot be applied to (Rational)
  2 * r
    ^
```

Implicit conversions

scala> implicit def intToRational(x: Int) = new Rational(x)

scala> val r = new Rational(2,3)
r: Rational = 2/3

scala> 2 * r
res16: Rational = 4/3
Exercises

1. Create a ComplexNum class that has two Double class parameters: real and imaginary.
2. Override toString to return “<real> + <imaginary>i”
3. Add an auxiliary constructor that takes a real part, assuming an imaginary part of 0.
4. Add a + method that takes a ComplexNum, and returns the sum of itself and the passed ComplexNum.
5. Add a + method that takes an Double, and returns the sum of itself and the passed Double.
4. Pattern Matching
abstract class Expr

case class Var(name: String) extends Expr

case class Number(num: Double) extends Expr

case class UnOp(operator: String, arg: Expr) extends Expr

case class BinOp(operator: String,
left: Expr,
right: Expr) extends Expr
Case classes: factory method

scala> val v = Var("x")
v: Var = Var(x)

scala> val op = BinOp("+", Number(1), v)
op: BinOp = BinOp(+,Number(1.0),Var(x))
Case classes: implicit vals

scala> v.name
res0: String = x

scala> op.left
res1: Expr = Number(1.0)
Case classes: natural toString, equals, and hashCode

scala> println(op)
BinOp(+,Number(1.0),Var(x))

scala> op.right == Var("x")
res3: Boolean = true
Match expressions

Scala:
selector `match` { alternatives }

Java:
`switch` (selector) { alternatives }
Constant and wildcard patterns

```scala
val firstArg = if (!args.isEmpty) args(0) else ""

val friend =
  firstArg match {
    case "salt" => "pepper"
    case "chips" => "salsa"
    case "eggs" => "bacon"
    case _ => "huh?"
  }

println(friend)
```
Constructor patterns

```scala
expr match {
  case BinOp("+", e, Number(0)) => println("a deep match")
  case _ =>
}
```
Variable patterns

expr match {
  case 0 => "zero"
  case somethingElse => "not zero: " + somethingElse
}


def tupleDemo(expr: Any) =
  expr match {
    case (a, b, c) => println("matched " + a + b + c)
    case _ =>
  }
Typed patterns (and placeholders)

def generalSize(x: Any) = x match {
  case s: String => s.length
  case m: Map[_, _] => m.size
  case _ => -1
}

javax.servlet.ServletRequest

public java.lang.String getParameter(java.lang.String name)

Returns the value of a request parameter as a String, or null if the parameter does not exist
Some(param) or None
scala.List

def find (p: (A) => Boolean): Option[A]

Find and return the first element of the list satisfying a predicate, if any.
val opt = 
  args.find(
    arg => { arg.substring(0, 2) == "-g" } 
  )

opt match {
  case Some(dashG) =>
    println("Found: " + dashG)
  case None =>
    println("No -g found")
}

Using Option
1. Create a function named `gimmeOptions` that takes a string, and returns the string wrapped in a `Some` if it begins with A through L (case insensitive), otherwise returns `None`.

2. Create a singleton object with a main method that passes each command line argument to the `gimmeOptions` method, and prints the result.

3. Use a pattern match in main that prints a string returned in `Some`, but for `None`, prints “Was not A to L”.

Exercises
Learn Scala

*Programming in Scala* is available in the Devoxx bookstore.