DZone Refcardz

CONTENTS INCLUDE:

- Downloading and Installing Scala
- Using the REPL
- Scala as a Scripting Language
- Using The Scala Compiler
- Language Features
- Collections...and More!

Scala

The Scalable JVM Language

By Jesper de Jong

WHAT IS SCALA?

Scala is a general-purpose programming language that has both object oriented and functional programming language features. It is statically typed, with an advanced type system that is more powerful than that of other languages such as Java and C#. The name Scala comes from scalable language: scalable in the sense that the core language constructs of Scala are so modular that new features can merely be added by libraries instead of adjusting the language. Scala is suitable for building large modular systems as well as for writing very small programs (using it as a scripting language). The functional features make it easy to write concurrent programs that make use of multiple cores or processors. Scala compiles to Java bytecode that runs on the Java virtual machine. It has very good interoperability with Java; Java libraries can be used seamlessly from Scala. Support for compiling Scala to .NET bytecode that runs on the CLR is work in progress.

DOWNLOADING AND INSTALLING SCALA

You can download Scala at www.scala-lang.org. To use Scala, you will need to have a Java 5 or newer runtime environment installed. It works on Windows, Linux, Mac OS X and other operating systems for which a suitable Java runtime environment is available. Follow the installation instructions for your operating system. If you install Scala from one of the available archive packages (a .zip or .tar.gz file), the only thing you need to do after unpacking the archive is add the bin directory of your Scala installation to the PATH environment variable.

USING THE REPL

Scala comes with an interactive interpreter, the REPL (for Read, Evaluate, Print Loop). After installing Scala, you can start the REPL by entering the command scala in a command prompt or shell window. It will look like this:

```
$ scala
Welcome to Scala version 2.9.1.final (Java HotSpot(TM) 64-Bit Server VM,
Java 1.7.0_02).
```

You can type statements and expressions at the prompt, and the REPL will evaluate these and print the results. The REPL is a useful tool for experimenting with Scala language features. The REPL understands a number of special commands that start with a colon. Type the command :help to see an overview. Type :quit to exit the REPL.

SCALA AS A SCRIPTING LANGUAGE

To use Scala as a scripting language, simply write Scala statements and expressions into a text file. Save the file and run it by passing the name of the file to the scala command.

```
// HelloScript.sh
// Run this with: scala HelloScript.sh
val name = readLine("What is your name? ")
println("Hello " + name + ", welcome to Scala!")
```

To run your script without writing the scala command in front of it, add a few special lines at the start of your script (a "shebang" in Unix terminology). Here is an example for Linux and Mac OS X:

```
#!/usr/bin/env scala
!#
val name = readLine("What is your name? ")
println("Hello " + name + ", welcome to Scala!")
```

Make the script executable by entering the command chmod u+x HelloAgain.sh and then run it with: ./HelloAgain.sh
Here is an example for Windows:

```
::#!
@echo off
scala %0 %*
goto :eof
::!#
val name = readLine("What is your name? ")
println("Hello " + name + ", welcome to Scala!")
```

For Windows, the script must have the extension .cmd or .bat. Otherwise, Windows will prompt you to open the file instead of running it as a script. Run the script by entering the command HelloAgain.cmd in the command prompt window.

The Scala interpreter will ignore anything between #! and !# (Linux and Mac OS X) or ::#! and ::!# (Windows) at the start of the script.

USING THE SCALA COMPILER

Compile Scala source files using the Scala compiler, scalac. When you use Scala as a compiled language your code must be organized a little differently than when using it as a scripting language. You'll need to create a Scala object with a main method, as in the following example.

```
// HelloWorld.scala
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello World!")
  }
}
```

Save this in a source file named HelloWorld.scala. Compile it with: scalac HelloWorld.scala and run it with: scala HelloWorld. Note that in this last command, you supply the name of the object (HelloWorld) rather than the name of the source file (HelloWorld.scala) or the compiled class file (HelloWorld.class).

Build tools

When you start creating projects with multiple source files, it quickly becomes cumbersome to build them all by hand with the scalac command. There are several build tools available to help you automate the build process, including Apache Ant, Apache Maven, or sbt (simple build tool, see www.scala-sbt.org).

IDE support

Several IDEs have support for Scala. There is an Eclipse-based Scala IDE that you can find at www.scala-ide.org. There are also Scala plug-ins available for JetBrains IntelliJ IDEA and NetBeans.

LANGUAGE FEATURES

Values and variables

```
val name: Type = expression
var name: Type = expression
```

With val, you define an immutable value (it cannot be reassigned after it is initialized). With var, you define a mutable variable. Identifiers are not limited to alphanumeric characters; many other characters can also be used, including Unicode mathematical and other symbols. Types are specified by a colon followed by the name of the type. The type can be omitted in most cases; Scala automatically finds out what the type must be through type inference.

```
val message = “Hello World”
val π: Double = 3.14159265358979
val result = 1 to 100 sum
```

You can also use pattern matching when defining values and variables.

```
// Defines x: Int and xs: List[Int], x is assigned the head
// and xs the tail of the list
val x :: xs = List(1, 2, 3, 4)

// A tuple with two Ints
val tup = (3, 5)

// Defines two Ints a and b and assigns them 3 and 5,
// the values extracted from the tuple
val (a, b) = tup
```

Types

The following table shows an overview of Scala's standard types.

Type	Description
scala.Any	The root of all Scala classes.
scala.AnyVal	The root of the value type classes (which map to JVM primitive types).
scala.AnyRef	The root of all reference type classes (analogous to java.lang.Object).
scala.ScalaObject	A trait inherited by all Scala classes and objects, but not by non-Scala classes (for example Java classes).
scala.Null	The type of the value null.
scala.Nothing	The type at the bottom of the class hierarchy.

The type Nothing is a special type that is at the bottom of the class hierarchy. This means that Nothing is a subtype of all other types. No instances of Nothing exist.

Similarly, the type Null is a subtype of all reference types (all types that inherit from AnyRef) except Nothing. In other words, Null is a subtype of everything that can have the value null.

The following table shows an overview of the subclasses of type scala.AnyVal. These map to Java primitive types on the JVM.

Type	Java	Description
scala.Byte	byte	8-bit signed integer.
scala.Short	short	16-bit signed integer.
scala.Int	int	32-bit signed integer.
scala.Long	long	64-bit signed integer.
scala.Char	char	16-bit unsigned integer used for Unicode characters.
scala.Float	float	Single-precision floating point number.
scala.Double	double	Double-precision floating point number.
scala.Boolean	boolean	Boolean value (true or false).
scala.Unit	void	Single value type.

The type Unit is similar to void in Java and other programming languages. There is a single instance of the type Unit which is written as ().

Tuples

Tuples are type-safe containers for multiple values. They are for example useful if you want to return more than one value from a method. They are different from collections; each element in a tuple has its own type, whereas in collections all elements in the collection usually have the same type.

```
// Defines a tuple with a String and an Int
val tup = (“Hello”, 123)

// The elements of a tuple are named _1, _2 etc.
println(tup._1)
println(tup._2)

// Method that returns a Tuple with two Ints
def div(a: Int, b: Int): (Int, Int) = (a / b, a % b)

// Call the method, use pattern matching to extract the values from the
// result, throw away the second value
val (x, _) = div(20, 7)
```

Scala's standard library has types Tuple2, Tuple3, up to Tuple22 to represent tuple types with up to 22 values.

Function types

Since Scala is a strongly typed functional programming language, functions are treated as values and functions also have a type. The type of a function that takes parameters of type T1, T2, ..., Tn and returns a value of type R is described with the following syntax:

```
(T1, T2, ..., Tn) => R
```

Here are some examples.

```
// A function that takes two Ints and returns an Int
val add: (Int, Int) => Int = { (a: Int, b: Int) => a + b }

// A function that takes an Int and returns a function
// that takes an Int and that returns Int
val times: Int => Int => Int = { (a: Int) => (b: Int) => a * b }

// Create a function to multiply numbers by 2
val timesTwo = times(2)

val n = timesTwo(21)
```

Scala's standard library has traits Function0, Function1, up to Function22 to represent the types of functions that take from 0 up to 22 parameters.

Option, Some and None

In Scala, you normally do not use null to indicate that a value is empty; instead, you use Option, Some and None. An Option is a container that may contain a value or be empty.

Some is a subclass of class Option that is used if the option contains a value.

None is an object that extends class Option that is used if the option is empty.

Option has a number of useful methods. Notably, it can be treated as a collection so that collection methods such as map, flatMap, filter, and foreach can be called on it. The most useful methods of class Option are listed below.

Method	Result if not empty	Result if empty
get	The option's value.	Throws a NoSuchElementException.
getOrElse	The option's value.	The specified default value.
isDefined	true.	false.
isEmpty	false.	true.
orElse	This Option.	The specified alternative Option.
exists	The result of applying the specified predicate on the option's value.	false.
filter	This Option if the result of applying the specified predicate on the option's value is true; None otherwise.	None.
filterNot	This Option if the result of applying the specified predicate on the option's value is false; None otherwise.	None.
foreach	Applies the specified procedure on the option's value; returns Unit.	Does nothing; returns Unit.
map	A Some containing the result of applying the specified function on the option's value.	None.
flatMap	The result of applying the specified function on the option's value. The function must return an Option.	None.
toList	A List containing one element, the option's value.	The empty list.

Expressions

Almost anything that would be considered a statement in other programming languages is an expression in Scala. The difference between an expression and a statement is that an expression results in a value that can be assigned to a variable or returned from a method and a statement does not result in a value. For example, if, for, and try are all expressions in Scala.

```
// The result of an if expression can be assigned to a val
val result = if (n % 2 == 0) n else 2 * n
```

For comprehensions

There is a flexible and powerful syntax for for loops. Between the braces that can be round or curly braces after the keyword for, you can include one or more generators, optionally with guards, that generate the values to loop over.

A generator looks like pattern <- expr where pattern is the name of a variable or a pattern matching expression. A guard after a generator is an if expression that filters values generated by the generator; if the guard expression for a generated value evaluates to false, then that value is skipped. A generator can have multiple guards. When you have multiple generators, it is the same as having nested for loops.

Between the braces of a for, you can also define values by specifying pattern = expr where pattern is a variable name or other pattern matching expression.

If the yield keyword is used after the braces that surround the generators, the for loop will return a collection containing the result of applying the body of the loop to the values that the loop iterates over.

```
// Returns a collection containing 0, 2, 4, ..., 18
val even = for (i <- 0 until 10) yield i * 2

// Loop with a value definition
val even = for (i <- 0 until 10; k = i * 2) yield k

// Generator with a guard
val even = for (i <- 0 until 20 if i % 2 == 0) yield i

// Loop with two generators and a value definition
for (x <- 1 to 10; y <- 1 to 10; p = x * y)
  printf("%d x %d = %d%n", x, y, p)
```

Methods

```
def name[TypeParams](ParamList1)(ParamList2)(...): ReturnType = {
  // Statements and expressions
}
```

With def inside a class, object, or trait, you define a method. Methods can optionally have type parameters and zero or more parameter lists. The return type can be omitted in most cases; Scala automatically finds out what the return type should be through type inference. If you omit the return type and the = between the method declaration and the method body, then the method returns Unit (like void in Java and other programming languages).

Hot Tip Make sure that you don't forget the = before the method body. If you do, the value that you intend to return from the method is lost.

The curly braces around the method body can be omitted if the body consists of a single expression. The return value of the method is the value of the last expression in the method body. Scala does have a return keyword, but you don't need it for most methods.

```
// A method with two parameter lists
def calc(a: Int, b: Int)(c: Int) = a * b + c

// A method that takes a 2-tuple with elements of type T
// and a function that takes a T and returns Unit
def forBoth[T](tup: (T, T), p: T => Unit) { p(tup._1); p(tup._2) }

val m = ("Hello", "World")
forBoth(m, println)
```

Default parameters

Parameters may have a default value that is used if a value is not supplied when calling the method.

```
def greet(name: String = "there") {
  println("Hello " + name)
}
```

Call-by-name parameters

If the type of a parameter is specified as => T (where T is the name of a type), then the parameter is a call-by-name parameter. Instead of being evaluated once before the method is called, the parameter will be evaluated every time it is used inside the method.

```
def time[T](block: => T): T = {
  val t0 = System.nanoTime()
  val result = block          // block will be evaluated here
  val t1 = System.nanoTime()
  println("Elapsed time: " + (t1 - t0) + " ns")
  result
}

val result = time { 1 to 1000 sum }
```

Repeated parameters

If the type of a parameter is followed by *, then the parameter is a repeated parameter. Only the last parameter in a parameter list can be a repeated parameter. Inside the method, the parameter will look like a scala.Seq.

```
def prettyConcat(names: String*) = {
  names.length match {
    case 0 => ""
    case 1 => names.head
    case _ => names.init.mkString(", ") + " and " + names.last
  }
}
```

When you want to pass the content of a Seq to a method with repeated parameters, append : _* after the parameter.

```
val names = List("Tom", "Susan", "Steve")
val s = prettyConcat(names: _*)
```

Calling methods and functions

There are two syntaxes for calling methods and functions: The regular syntax where the name of a value or variable is followed by a dot, the name of the method or function and the parameters between round braces, or operator-style syntax. Any method or operator can be called with either of these two syntaxes.

Operators

Operators are defined just like regular methods. Method names are not limited to alphanumeric characters, so you can use characters that would normally be used for operators as method names. A limited number of operators can be used as prefix operators in expressions that have the form op expr. The operators can be used as prefix operators are +, -, !, and ~.A prefix operator expression op expr is equivalent to a regular method call expr.unary_op. This means that you can define your own prefix operator by defining a method named unary_op.

```
case class Example(n: Int) {
    def unary_- = new Example(-n)
}

scala> val a = new Example(57)
a: Example = Example(57)

scala> val b = -a
b: Example = Example(-57)
```

All methods that take no parameters can be used as postfix operators in expressions that have the form expr op. Such an expression is equivalent to expr.op.

For infix operators where the expression has the form expr1 op expr2, there are rules to determine the operator precedence and associativity. The precedence is determined by the first character of the name of the operator. The following table gives an overview in order of decreasing precedence.

Operator precedence by first character of the operator name
(all other special characters)
* / %
+ -
:
= !
< >
&
^
|
(all letters)

There is one exception: the assignment operator = has lowest precedence.

The associativity of an operator is determined by the last character of the name of the operator. If this is a :, then the operator is right-associative; otherwise it is left-associative. In other words, if the operator name does not end with :, then the expression expr1 op expr2 is equivalent to expr1.op(expr2); otherwise it is equivalent to expr2.op(expr1) (or, more formally, {val tmp = expr1; expr2.op(tmp)} because the first expression is evaluated first).

Partial function application

Functions can be partially applied, which means that you fill in some but not all of the parameters for the function. This results in another function with fewer parameters.

```
val times: (Int, Int) => Int = { (a: Int, b: Int) => a * b }

// Use an underscore to indicate that the parameter
// is not yet filled in; timesTwo will be a function
// of type Int => Int
val timesTwo = times(2, _: Int)

scala> timesTwo(3)
res0: Int = 6
```

Functions

The difference between a function and a method is that a function is a value (an object) while a method is not. A function is an instance of one of the traits Function0, Function1 etc., depending on the number of parameters that the function takes. Calling a function is equivalent to calling the apply method on the function object.

```
// succ is an instance of Function1
val succ: Int => Int = { _ + 1 }

// Calling a function is equivalent to calling its apply method
succ(3)
succ.apply(3)
```

You can convert a method to a function by assigning it to a val or var. However, to avoid ambiguity, you have to either explicitly specify the type of the val or var, or put an underscore after the method name to indicate that you want to treat it as a function.

```
// A method
def succ1(x: Int) = x + 1

// Explicitly specify the type of succ2
val succ2: Int => Int = succ1

// Or put an underscore after the method name
val succ2 = succ1 _
```

Classes

```
class Name[TypeParams] AccessModifier (ParamList1)(ParamList2)(...)
  extends ClassOrTrait with Trait1 with Trait2 with ... {
    // Constructor statements and class members
  }
```

Classes can optionally have type parameters. AccessModifier is an optional access modifier (private or protected, optionally with a qualifier) that determines the accessibility of the primary constructor of the class. The primary constructor can have zero or more parameter lists. Parameters can be prefixed with val or var, in which case the parameters become class members.

A class can extend one other class and multiple traits. If the extends clause is omitted, the class automatically extends scala.AnyRef.

The body of the class can also be omitted; this is equivalent to including an empty body {}.

Constructors

The parameters for the primary constructor are specified in the header of the class definition before the class body. Any statements inside the class body are executed when a new instance of the class is initialized as part of the primary constructor.

```
class Greeter(name: String) {
  // Statement executed as part of the primary constructor
  println("New Greeter: " + name)

  def greet() {
    println("Hello, " + name)
  }
}
```

You can specify additional constructors with def this. The first statement inside an additional constructor must be a call of the form this(...) to a previously defined additional constructor or to the primary constructor. This means that when a new instance is initialized, ultimately the primary constructor will always be called.

```
class Greeter(name: String) {
  // Statement executed as part of the primary constructor
  println("New Greeter: " + name)

  // Additional constructor
  def this(name1: String, name2: String) {
    // Call to primary constructor
    this(name1 + " and " + name2)
  }

  // Additional constructor
  def this(names: String*) {
    // Call to previously defined additional constructor
    this(names.init.mkString(", "), names.last)
  }

  def greet() {
    println("Hello, " + name)
  }
}
```

Case classes

A case class is similar to a regular class but has a number of special rules. The parameters of the first constructor parameter list automatically get a

val prefix (unless you explicitly added a var prefix). In other words, a getter is automatically added to the class.

A copy method is automatically added to the class.

The equals, hashCode, and toString methods are automatically overridden with appropriate implementations.

An apply and an unapply method for pattern matching are automatically added to the companion object of the class. (When the class is abstract, the apply method is omitted).

If the first constructor parameter list ends with a repeated parameter, an unapplySeq method instead of an unapply method is added to the companion object.

Case classes are mainly intended for pattern matching, but they are also often used for convenience to avoid having to implement methods such as equals, hashCode and toString manually.

```
scala> case class Person(name: String, age: Int)
defined class Person

scala> val p = Person("Sarah", 50)
p: Person = Person(Sarah,50)
```

Objects

Scala has no static keyword. Instead, you create singleton objects with the object keyword.

```
object Name extends ClassOrTrait with Trait1 with Trait2 with ... {
  // Statements and object members
}
```

Any statements inside the object's body are executed when the object is first used (as if they are in a static initializer in Java).

An object can extend one class and multiple traits. If the extends clause is omitted, the object automatically extends scala.AnyRef.

Companion classes and objects

If you define a class and an object with the same name in the same scope and source file, then the object is the companion object of the class, and the class is the companion class of the object. Companion classes and objects can access each other's private members.

Traits

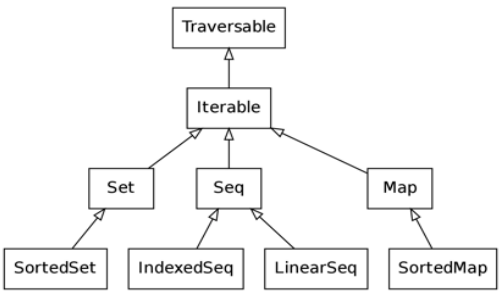
```
trait Name[TypeParams] extends ClassOrTrait with Trait1 with Trait2
with ... {
  // Constructor statements and trait members
}
```

A trait can extend one class and multiple other traits. A trait can have a constructor (you can put statements in the body of the trait that are executed as part of its constructor), but it cannot have constructor parameters.

Traits are like Java interfaces, except that a trait can be partially implemented. Like a Java interface, a trait cannot be directly instantiated even if it is fully implemented.

COLLECTIONS

Scala has a powerful and well-designed collections library. There are mutable and immutable versions of most collections. The following image shows an overview of the basic collection traits.



Collection	Description
Traversable	The top-level collections trait; declares the foreach method and defines other methods that are available on all collections.
Iterable	Declares the iterator method; all collections that extend Iterable can provide an iterator.
Seq	The base trait for sequences; a sequence has a defined order of elements.
IndexedSeq	The base trait for array-like sequences.
LinearSeq	The base trait for linked list-like sequences.
Set	The base trait for sets; a set contains no duplicate elements.
SortedSet	The base trait for sets that have a defined order of elements.
Map	The base trait for maps.
SortedMap	The base trait for maps that have a defined order of keys.

For each of these traits there are one or more mutable and immutable implementations available. The most important ones are listed below.

Collection	Immutable implementations
IndexedSeq	Vector NumericRange Range Array String
LinearSeq	List Queue Stack Stream
Set	HashSet ListSet BitSet
SortedSet	TreeSet
Map	HashMap ListMap
SortedMap	TreeMap

Collection	Mutable implementations
IndexedSeq	ArraySeq ArrayBuffer StringBuilder
LinearSeq	MutableList LinkedList DoubleLinkedList Queue
Set	HashSet LinkedHashSet BitSet
Map	HashMap LinkedHashMap ListMap

Instantiating collections

The collection traits and classes have companion objects with apply methods, which allow you to create instances in a uniform way. For the traits, an appropriate default implementation is returned.

```
Traversable(1, 2, 3)    // returns a List[Int]
Seq(1, 2, 3)            // returns a List[Int]
IndexedSeq(1, 2, 3)     // returns a Vector[Int]
```

Collection operations

The following is a list of the most common operations that are defined in the trait Traversable; they are available on all collections.

Operation	Description
xs foreach f	Evaluates the function f for every element of xs.
xs ++ ys	Returns a collection that contains the elements of both xs and ys.
xs map f	Applies the function f to all elements and returns a collection with the transformed elements.
xs flatMap f	Applies the function f to all elements; f returns a collection, flatMap concatenates all these collections and returns the result.
xs collect f	Applies the partial function f to all elements where f is defined and returns a collection with the transformed elements. This is a filter and map operation in one; equivalent to xs filter f.isDefined map f.
xs.isEmpty	Returns true if the collection does not contain any elements, false otherwise.
xs.nonEmpty	Returns true if the collection contains at least one element, false otherwise.
xs.size	Returns the number of elements.
xs.hasDefiniteSize	Returns true if the collection is known to have finite size. Some collections, such as streams, do not always have a finite size.
xs.head	Returns the first element of the collection; throws an exception if the collection is empty.

xs.headOption	Returns a Some with the first element of the collection or None if the collection is empty.
xs.last	Returns the last element of the collection; throws an exception if the collection is empty.
xs.lastOption	Returns a Some with the last element of the collection or None if the collection is empty.
xs.tail	Returns a collection with all but the first element of this collection; throws an exception if the collection is empty.
xs.init	Returns a collection with all but the last element of this collection; throws an exception if the collection is empty.
xs take n	Returns a collection with the first n elements of this collection.
xs drop n	Returns a collection without the first n elements of this collection.
xs takeWhile p	Returns a collection with the longest prefix of this collection with elements for which p returns true.
xs dropWhile p	Returns a collection without the longest prefix of this collection with elements for which p returns true.
xs find p	Returns a Some with the first element for which p returns true or None if there is no such element.
xs filter p	Returns a collection with all elements for which p returns true.
xs filterNot p	Returns a collection with all elements for which p returns false.
xs withFilter p	Returns a non-strict filter on this collection that shows all elements for which p returns true.
xs slice (from, to)	Returns a subcollection of xs from index from up to and excluding index to.
xs splitAt n	Splits xs at index n, returning the pair of collections (xs take n, xs drop n).
xs span p	Splits xs using the predicate p, returning the pair of collections (xs takeWhile p, xs dropWhile p).

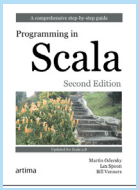
xs partition p	Splits xs using the predicate p, returning the pair of collections (xs filter p, xs filterNot p).
xs groupBy f	Returns a map of collections; f takes an element and returns the key under which that element must be stored in the map.
xs forall p	Returns true if p returns true for all elements; false otherwise.
xs exists p	Returns true if p returns true for at least one element; false otherwise.
xs count p	Returns the number of elements for which p returns true.
xs.foldLeft(z)(op)	Applies the operation op between successive elements, going left to right and starting with the value z.
(z /: xs)(op)	Alternative name for foldLeft.
xs.foldRight(z)(op)	Applies the operation op between successive elements, going right to left and starting with the value z.
(xs :\ z)(op)	Alternative name for foldRight.
xs reduceLeft op	The same as foldLeft but takes the first element of the collection as the initial value. Throws an exception if the collection is empty.
xs reduceRight op	The same as foldRight but takes the last element of the collection as the initial value. Throws an exception if the collection is empty.
xs.sum	The sum of the numeric elements in xs.
xs.product	The product of the numeric elements in xs.
xs.min	The minimum of the ordered elements in xs.
xs.max	The maximum of the ordered elements in xs.

ABOUT THE AUTHOR

Jesper de Jong is an experienced self-employed software engineer who has been working with Java since 1998. He designs and implements scalable, highly concurrent, mission-critical business systems for clients in different market sectors. Scala has been one of his interests since 2008 and he is active as a member of the Dutch Scala Enthusiasts user group in the Netherlands. Jesper has a blog about programming in Scala at www.scala-notes.org. He can be reached at jesper@jdj-it.com.

Thanks to Urs Peter, Age Mooij, Cay Horstmann and the DZone community for reviewing this refcard.

RECOMMENDED BOOK



Written by the designer of the language, Martin Odersky, Co-authored by Lex Spoon and Bill Venners. This book takes a step-by-step tutorial approach to teaching you Scala. Starting with the fundamental elements of the language, Programming in Scala introduces functional programming from the practitioner's perspective, and describes advanced language features that can make you a better, more productive developer.

Buy Here

DZone, Inc.
150 Preston Executive Dr.
Suite 201
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-936502-59-2
ISBN-10: 1-936502-59-3
50795
9 781936 502592
$7.95

Version 1.0