Scala for Java Developers

The Scalable Language for Everyone

Matt Sicker
<http://musigma.org/>

SPR Consulting

11 July 2017

Outline

1. Scala Syntax
   - Expressions
   - Classes
   - Implicits

Outline

1. Scala Syntax
   - Expressions
   - Classes
   - Implicits

Variables

- Use **val** for unmodifiable variables
- Use **var** for modifiable variables
- There are very little restrictions on allowed characters in identifiers
- Use the _ variable as a throwaway variable

```
var foo = "bar" // foo is inferred to be a String
foo = "baz" // can reassign vars
```

Types

- Types come after the variable name
- Variable types can oftentimes be inferred by the compiler
- All types descend from Any and are all supertypes of Nothing
- The AnyRef class is an alias for java.lang.Object
- The AnyVal class describes value-type classes such as the primitive types

```
val name: String = "Matt"
```

Tuples

- Tuples are generalized ordered pairs
- Syntax sugar for instances of the TupleN traits

-
  ```scala
  val foo: (String, Int, Boolean) =
    ("Hello", 42, true)
  ```
- Can access individual parts using the _N methods or via destructuring:

-
  ```scala
  val (s, _, _) = foo
  val t = foo._2
  // s = "Hello", t = 42
  ```

Methods

- Methods are defined with **def**
- Can omit the return type if it's inferable
- Must return something or Unit
- Can define methods inside methods
- Can omit parenthesis for 0-arg methods
- Using **return** is usually optional

```
def greet(name: String): Unit = {
  def greeting(name: String): String =
    s"Hello,␣$name" // string templates
  println(greeting(name))
}
```

Lambda Functions

- Similar syntax to **def**, though without a method name, and with an arrow
- Very similar syntax to Java, but replace −> with **=>**
- Unlike Java, Scala lambdas can close over mutable variables
- Using **return** in a lambda will return from the outer named function, not the lambda
- Syntactical sugar for an anonymous class of FunctionN traits
- In Scala 2.12, lambdas were updated to be compatible with Java 8 lambdas

Lambda Example

```
val add = (a: Int, b: Int): Int => a + b
// equivalent to:
val add = new Function2[Int, Int, Int] {
  override def apply(a: Int, b: Int): Int =
    a + b
}
val sum = add(1, 2)
// equivalent to
val sum = add.apply(1, 2)
```

Conditionals

- An **if** expression returns the last value of the matching branch similar to the ternary operator in Java

-
  ```
  def describe(n: Int): String =
    if (n % 2 == 0) "even" else "odd"
  ```
- Can still perform side effects and return Unit

Loops

- Use **while** (expr) { expr } for simple loops
- Use **for** in a foreach loop or to transform collections using **yield**
- Can combine with ranges to get an indexed for loop

```
val langs = List("Java", "Scala", "Clojure")
val lower = for (lang <- langs)
  yield lang.toLowerCase
// equivalent to:
val lower = langs.map(lang => lang.toLowerCase)
```

Pattern Matching

- An expression can be matched in many ways:
- Type of expression
- Value of expression
- Types or values within the expression
- Uses **match** and **case**
- Additional predicates using **if**
- Similar to a switch statement in Java
- Protip: a block made up of **case** expressions is an anonymous match and can be used as a single-argument lambda function

Pattern Matching Example

```
def describe(x: Any): String =
  x match {
    case null => "null"
    case i: Int => i.toString
    case s: String if s.nonEmpty => s
    case Some(y) => describe(y)
    case None => "none"
    case _ => "unknown" // default case
  }
```

Exceptions

- All exceptions are unchecked in Scala
- Throw an exception with throw
- Catch exceptions using **try** and **catch**
- A **catch** block is a pattern match expression on the exception

```
def open(file: String) =
  throw new Exception("File␣not␣found")
try {
  val f = open("foo.txt")
} catch {
  case e: Exception =>
    println(e.getMessage)
}
```

Outline

1. Scala Syntax
   - Expressions
   - Classes
   - Implicits

Traits

- A **trait** is similar to an interface in Java
- Defines abstract methods and fields
- Can also define concrete methods and fields
- Very similar to an abstract class but cannot have a constructor
- Classes can inherit from multiple traits, but only from one class
- Traits can be restricted to certain implementing types

Trait Example

```
trait Logger {
  def isEnabled(level: String): Boolean
  def append(level: String, message: Any): Unit
  def log(level: String, message: Any): Unit =
    if (isEnabled(level)) append(level, message)
  def error(message: Any): Unit =
    log("ERROR", message)
  def debug(message: Any): Unit =
    log("DEBUG", message)
}
```

Classes

- A **class** works rather similarly to Java classes
- Contains a main constructor and optionally other constructors named **this**
- The body of the class (minus any **def**s) is the constructor
- Fields can be exposed using **val** and **var** for read-only and read-write properties
- Can extend another class and several traits
- Use **extends** to extend a class or implement a trait
- Use **with** to add additional traits to mix in to the class
- Use **override** on methods and variables overridden from parent
- Use **lazy val** for values that aren't evaluated until first access

Class Example

```
class StdoutLogger(levels: Map[String, Boolean])
    extends Logger {
  override def isEnabled(level: String): Boolean =
    levels(level)
  // marking as final prevents subclasses
  // from overriding
  final override def append(
      level: String, message: Any): Unit =
    println(s"$level:␣$message")
}
val logger = new StdoutLogger(
  Map("ERROR" -> true, "DEBUG" -> false))
```

Objects

- Scala does not have a static keyword
- It does however have a singleton **object** keyword
- An object is a class with only a single instance
- When named the same as a class, provides similar semantics to having static methods defined on the class itself

Object Example

```
object Logger {
  def apply(debug: Boolean, error: Boolean): Logger
    new StdoutLogger(Map(
      "DEBUG" -> debug,
      "ERROR" -> error
    ))
}
val logger = Logger(false, true)
logger.debug("Test")
```

Case Classes

- In Java, there is a lot of boilerplate to create a simple data class
- Using Lombok, we can avoid most of it by adding annotations like @Data, @Wither, @Builder, etc., to the class
- In Scala, we can add **case** to a **class** to get similar functionality generated for us: toString, equals, hashCode, copy, apply, unapply, Scala-style getters for the constructor parameters, an all-args constructor, and some other goodies

Case Class Example

```
case class Name(first: String, last: String)
// automatic Name.apply created
val john = Name("John", "Doe")
// automatic Name.unapply created
val Name(first, last) = john
def isAnon(n: Name): Boolean = n match {
  case Name(_, "Doe") => true
  case _ => false
}
// automatic Name.copy like @Wither
val jane = john.copy(first = "Jane")
```

Generics

- Unlike Java, Scala does not allow raw types
- Generic syntax uses square brackets instead of angled brackets
- For consistency, arrays use the same syntax as collections
- `val xs: Array[Int] = Array(1, 2, 3)`
- Can specify how instances relate using generic type parameter by specifying the variance (similar to super/extends in Java)
- Can use type parameter bounds using **>:** and **<:** for superclass and subclass respectively

Generic Example

```
// +A: if B extends A, then Bag[B] extends Bag[A]
trait Bag[+A] {
  // if B super A, then we widen the type
  def add[B >: A](b: B): Bag[B]
  def remove(p: A => Boolean): Bag[A]
  // defining map, flatMap, and foreach allow this
  // class to be used in various for expressions
  def map[B](f: A => B): Bag[B]
  def flatMap[B](f: A => Bag[B]): Bag[B]
  def foreach(f: A => Unit): Unit
}
```

Class and Method Parameters

- The arguments to the primary constructor of a class can be considered the class's arguments similar to a method
- Arguments can be passed by name out of order, negating the need for builders:

```scala
val log = Logger(error = true, debug = false)
```

- Parameters can have a default value:

```scala
def greet(name: String = "World") =
  s"Hello,␣$name"
val greeting1 = greet()
val greeting2 = greet("Chicago")
```

Repeated Parameters

- Similar to varargs in Java, the last argument in a parameter list can be a repeated parameter
- Access the variable as a Seq[T] collection class
- Expand a collection class to a repeated parameter using `: _*` on the variable

```
def join(fields: String*): String =
  fields.mkString(",")
val csv1 = join("foo", "bar", "baz")
val csv2 = join(List("foo", "bar"): _*)
```

Multiple Parameter Lists

- A class or method can contain multiple parameter lists
- This syntax can be useful for partial function application

```
trait Foldable[A] {
  def fold[B](init: B)(op: (B, A) => B)
}
val f: Foldable[Int] = ...
f.fold(0)((sum, next) => sum + next)
```

Outline

1. Scala Syntax
   - Expressions
   - Classes
   - Implicits

Implicit Conversions

- In many projects, common boilerplate code to convert from one type to another
- In Scala, we can create an implicit conversion to automatically convert types where applicable
- Implicits are a core feature of Scala that differentiate it from other languages

```
def foo(id: UUID): Unit = ...
implicit def s2id(s: String): UUID =
  UUID.fromString(s)
val id = "cea3e50d-f894-47fe-a31b-0cb57c94bea5"
foo(id)
// expands to:
foo(s2id(id))
```

Implicit Classes

- In order to add methods to third party classes, we can wrap the class and provide new methods
- Combined with an implicit conversion, we can use a shorthand syntax to make an implicit class
- An implicit class is a class with a single parameter with a generated implicit function to convert from the type of the parameter to the implicit class

```
implicit class IntOps(i: Int) {
  def isEven: Boolean = i % 2 == 0
}
val q = 42.isEven
```

DSL Example

```scala
case class Module(group: String, module: String)
implicit class Group(group: String) {
  def %(artifact: String): Module =
    Module(group, artifact)
}
val module = "org.apache.commons" % "commons-lang3"
```

Implicit Parameters

- Passing the same contextual information over and over again is repetitive
- Using implicit parameters along with implicit values helps reduce boilerplate

```
def fetch(query: String)
  (implicit conn: Connection): Seq[Row] =
  conn.query(query)
implicit val c: Connection = ...
val rows = fetch("select␣*␣from␣things")
// expands to:
val rows = fetch("select␣*␣from␣things")(c)
```

Implicit Context Bounds

- Some types are used to provide context for another type such as Ordering[T] for defining an ordering on a type
- We can provide context objects via implicit parameters

```
def min[A](a: A, b: A)
  (implicit o: Ordering[A]): A =
  o.min(a, b)
// or we can add a context bound
// and summon the implicit:
def min[A: Ordering](a: A, b: A): A =
  implicitly[Ordering[A]].min(a, b)
```

Summary

- Scala provides a small, consistent core language with lots of optional syntax sugar
- Works well with existing Java libraries
- Eliminates a lot of common boilerplate
- Java is slowly adopting old Scala features (lambda functions, streams, and eventually pattern matching), so why wait?
- This is only the basics; Scala provides a standard library with very rich collection classes, more syntax sugar for functional programming, and many production-ready libraries and frameworks

Further Reading

- http://musigma.org/scala/2017/07/03/akka-cqrs.html
- https://github.com/jvz/akka-blog-example
- https://github.com/jvz/scala-for-java