Scala

Overview

Short introduction / basic syntax
- If / else / for
- expressions
- closures / function literals
- Pattern matching

Object Oriented Programming
- Classes / inheritance / encapsulation
- Traits and interfaces
- Nested classes

Functional Programming
- Higher order functions
- Collections

Things I didn’t talk about

Additional resources

What is Scala?

- Statically typed, modern programming language for the JVM
- Seamless integration with Java libraries

Who's using Scala

- Twitter
- Airbnb
- LinkedIn
- Netflix
- Tumblr
- The Guardian
- Sony

- (check out Twitter and Netflix on GitHub)

Short Intro / basic syntax

Variables and expressions

```
// Basic declaration of a variable
var x: Int = 7

// Type inference
var y = 7

// Simple if test
if (y > 10) {
  x = 5
}

// Constant values
val z = 3

// Everything is an expression
val hello = if (x > 8) "Hello" else "World"
```

```
// Lazy evaluation
lazy val bigH = "Hello".charAt(0)

// Value declarations can span multiple lines
val hello2 = {
  val hello = "Hello, "
  val world = "World"
  hello + world
}
```

Functions

```
// Simple function declaration
def greet(name: String): String = {
  "Hello, " + name
}

// Compiler can infer return types
def greet2(name: String) = {
  "Hello, " + name
}

// No need for redundant brackets
def greet3(name: String) = "Hello, " + name
```

Functions as values

```
val greet: String => String = (name: String) => "Hello, " + name
                              name => "Hello, " + name
                              "Hello, " + _

greet("World") // Hello, World
```

```
// Can span multiple lines
val greet: String => String = { name =>
  val hello = "Hello, "
  hello + name
}
```

Operators

In Scala operators are regular method calls

```
// Equivalent
1 + 5
1.+(5)

// Equivalent
"Hello".charAt(5)
"Hello" charAt 5
```

Pattern matching

```scala
val number = 5

val str = number match {
  case 1 => "One"
  case 2 => "Two"
  case _ => "Not one or two"
}
```

```scala
// With guards
val fancy = number match {
  case 1 => "One"
  case 2 => "Two"
  case x if x > 10 => "HUGE"
  case x => "Number is " + x
}
```

```scala
val conclusion = someValue match {
  case x: Int => "x is an Integer"
  case x: String => "x is a String"
  case _ => "x is unknown"
}
```

```scala
// Custom extractors
val Email = """(\w+)@([\w\.]+)""".r

"test@example.com" match {
  case Email(name, domain) => println(domain)
  case _ => println("Not an email address.")
}
// Prints example.com
```

Object Oriented Programming

Classes

```scala
class Student(var name: String)

val student = new Student("Alex")


// Compiler generates:
//  def name: String
//  def name_=(param: String)

// Equivalent (operator syntax)
student.name = "Mr. Test"
student.name_=("Mr. Test")

import student._
name = "Test"
```

```scala
class Student(var name: String) {
  def nameAsUpperCase = name.toUpperCase
}
```

```scala
class Student(val name: String) {
  val nameAsUpperCase = name.toUpperCase
}
```

```scala
class Student(val name: String) {
  lazy val nameAsUpperCase = name.toUpperCase
}
```

Objects

```scala
class Student(var name: String)

object Student {
  def apply(name: String) = new Student(name)
}
```

```scala
val student = Student.apply("Alex")
```

```scala
val student = Student("Alex")
```

Traits: As an interface

```scala
trait Person {
  def name: String
}

class Student(val name: String) extends Person
```

Traits: As an interface with a default implementation (mixin)

```
trait Grades {
  def grades: Map[String, String]

  def printGrades() = {
    for ((course, grade) <- grades) {
      println(s"$course\t$grade")
    }
  }
}

class Student(val name: String, val grades: Map[String, String])
  extends Person with Grades

val grades = Map(
  "Some Course 101" -> "A",
  "Some Other Course" -> "A"
)

val student = new Student("Mr. Good", grades)

student.printGrades()
// Some Course 101          A
// Some Other Course        A
```

Pattern matching on traits

```scala
class Student(val name: String)

trait Grades {
  def grades: Map[String, String]

  def printGrades() = {
    for ((course, grade) <- grades) {
      println(s"$course\t$grade")
    }
  }
}
```

```scala
val mrGood =
  new Student("Mr. Good") with Grades {
    val grades = Map(
      "Some Course 101" -> "A",
      "Some Other Course" -> "A"
    )
  }

val mrNew = new Student("Mr. New")
```

```scala
def printGradesIfAvailable(student: Student) = student match {
  case s: Grades => s.printGrades()
  case s: Student => println(s.name + " has no grades.")
}
```

```
printGradesIfAvailable(mrGood)
Some Course 101    A
Some Other Course  A

printGradesIfAvailable(mrNew)
Mr. New has no grades.
```

Case Classes

```
trait Person {
  def name: String
}

case class Student(name: String, age: Int) extends Person
case class Professor(name: String) extends Person


def whoAmI(who: Person) = who match {
  case Student(name, age) => "You're a student, and your name is " + name
  case Professor(name) => "You're professor " + name
  case person => "Your name is " + person.name
}


val student = Student("Mr. Good", 19)
val professor = Professor("Awesome")

whoAmI(student)    //=> res0: String = You're a student, and your name is Mr. Good
whoAmI(professor)  //=> res1: String = You're professor Awesome
```

Case Classes

```
sealed trait Tree
case class Node(left: Tree, right: Tree) extends Tree
case class Leaf(name: String) extends Tree


val tree: Tree = Node(
  Node(Leaf("a"), Leaf("b")),
  Node(Leaf("c"), Leaf("d"))
)


def findValues(tree: Tree): List[String] = tree match {
  case Leaf(value) => value :: Nil
  case Node(left, right) => findValues(left) ::: findValues(right)
}


findValues(tree)
res2: List[String] = List(a, b, c, d)
```

Functional Programming

Higher Order Functions

Functions that

- take other functions as parameters -or-
- returns another function as a return value -or-
- both

Use cases ...

Higher Order Function

Use Case: Simplifying the usage of locks

```
val lock = new Lock
var x: String = ""

x = try {
  lock.acquire()
  "Hello, World"
} finally {
  lock.release()
}
```

```
def synchronize[T](f: => T): T = {
  try {
    lock.acquire()
    f
  } finally {
    lock.release()
  }
}

x = synchronize {
  "Hello, World"
}
```

Already exists in Scala, and is part of every object.
It's called `synchronized`.

Lists

```scala
val list1 = List(1,2,3,4,5,6,7)


// Alternative:
val myList = "This" :: "is" :: "a" :: "list" :: Nil


// Can pattern match on lists
def listMatch(list: List[String]): Unit = list match {
  case "Magic" :: tail =>
    println("Magic! :)")
  case first :: second :: tail =>
    println("First item is %s, second item is %s".format(first, second))
  case _ =>
    println("List with 1 item")
}


listMatch(List("Magic")) //=> Magic! :)
listMatch(List("hello", "world")) //=> First item is hello, second item is world
listMatch(List(":)")) //=> List with 1 item
```

Transformations

```scala
val myList = "This" :: "is" :: "a" :: "list" :: Nil

val wordLengths = myList.map(word => word.length)
        //=> List[Int] = List(5, 5, 4, 2, 1, 4)

val characters = myList.flatMap(word => word.toList)
        //=> List[Char] = List(T, h, i, s, i, s, a, l, i, s, t)

val wordsShorterThanThree = myList.filter(word => word.length < 3)
        //=> List[String] = List(is, a)

val characters1: List[Char] =
  myList.filter(word => word.length < 3) //=> List[String] = List(is, a)
    .flatMap(word => word.toList)        //=> List[Char] = List(i, s, a)

val characters2: List[Char] =
  for {
    word <- myList
    if word.length < 3
    character <- word
  } yield character
```

`for` compiles to a bunch of
- map()
- flatMap()
- withFilter() // similar to filter

`for` doesn't care about the implementation!

Transformations

Task: Find every line in a Wikipedia article that contains the word "ship", calculate it's length, and compute the sum of the lengths.

```
val text =
  """USS Lexington (CV-2), nicknamed "Lady Lex",[1] was an
    | early aircraft carrier built for the United States Navy.
    | She was the lead ship of the Lexington class; her only
    | sister ship, Saratoga, was commissioned a month earlier.""".stripMargin

val lines: List[String] = text.split("\n").toList
val linesWithShip: List[String] = lines.filter(line => line.contains("ship"))

val lineLengths: List[Int] = linesWithShip.map(line => line.length)

val total: Int = lineLengths.reduce((lhs, rhs) => lhs + rhs)
```

Alternative

```
val lineLengths: List[Int] =
  for {
    line <- lines
    if line.contains("ship")
  } yield line.length

val total = lineLengths.reduce(_+_)
```

Transformations: Distributed version

```
val sc = new SparkContext(master, appName, sparkHome, jars)
val lines: RDD[String] =
  sc.textFile("hdfs://compute-0-0/user/ira005/" +
    "exampleData/wikipedia/enwiki-latest-pages-articles.xml")
    .cache()
```

```
val lineLengths: RDD[Int] =
  for {
    line <- lines
    if line.contains("ship")
  } yield line.length

val total: Int = lineLengths.reduce(_+_)
```

From previous slide:

```
val lineLengths: List[Int] =
  for {
    line <- lines
    if line.contains("ship")
  } yield line.length

val total = lineLengths.reduce(_+_)
```

Time of iteration 1 (ms): 95360
Time of iteration 2 (ms): 2308
Time of iteration 3 (ms): 2177
Time of iteration 4 (ms): 2167

Futures

- Placeholder for a value that will be available at some point in the future (or an exception in the case of failure)
- Lets you register callbacks and perform transformations on the future value
- Many different implementations

Futures (com.twitter.util.Future)

```
def httpClient(host: String): Request => Future[Response]

val client1 = httpClient("localhost:8000")
val request = Request()
request.uri = "/"

val responseFuture1: Future[Response] = client1(request)

// Simplistic: Wait for result (Blocks until ready. Will throw on error.)
val response: Response = Await.result(responseFuture1)

    def map[B](f: A => B): Future[B]

// Alternative: Transform the future (doesn't block)
val statusCodeFuture: Future[Int] =
  responseFuture1.map { response: Response =>
    response.statusCode
  }

 statusCodeFuture.onSuccess(code => println(code))
                 .onFailure(e    => e.printStackTrace())
```

Futures (com.twitter.util.Future)

```
val client1 = httpClient("localhost:8000")
val responseFuture1: Future[Response] = client1(request)

val client2 = httpClient("localhost:8000")
val responseFuture2: Future[Response] = client2(request)
```

```
def flatMap[B](f: A => Future[B]): Future[B]
```

```
val codesFuture: Future[(Int, Int)] =
  responseFuture1.flatMap { resp1 =>
    responseFuture2.map { resp2 =>
      (resp1.statusCode, resp2.statusCode)
    }
  }
```

```
val codesFuture: Future[(Int, Int)] =
  for {
    response1 <- responseFuture1
    response2 <- responseFuture2
  } yield {
    val code1 = response1.statusCode
    val code2 = response2.statusCode
    (code1, code2)
  }
```

```
codesFuture.onSuccess( codes => println(codes) )
```

Questions