Um tutorial de Scala
para programadores Java

Versão 1.2
18 de setembro de 2008

Michel Schinz
Philipp Haller

Tradução:
Marcelo Castellani
Thiago Rocha

PROGRAMMING METHODS LABORATORY
EPFL
SWITZERLAND

1 Introdução

Este documento visa dar uma rápida introdução a linguagem Scala e seu compilador. Ele é dirigido a pessoas que realmente possuem alguma experiência em programação e desejam uma visão geral da linguagem Scala. Assumimos que você possui conhecimento básico em conceitos de orientação a objeto, especialmente em Java.

2 Um primeiro exemplo

Como primeiro exemplo usaremos o clássico *Hello world*. Ele não é um programa fascinante, mas com ele fica simples de demonstrar o uso de ferramentas Scala sem falar muito da linguagem. Ele ficará como abaixo:

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

A estrutura deste programa deve ser familiar para programadores Java: ela consiste de um método chamado `main` que recebe os argumentos da linha de comando, um array de string, como parâmetro; o corpo deste método consiste de uma chamada única ao método pré-definido `println` que recebe nossa amigável saudação como argumento. O código de `main` não retorna um valor, dessa forma não é necessário declarar um valor de retorno.

O que não é tão familiar assim para programadores Java é a declaração **`object`** que contém o método `main`. Essa declaração introduz o que é comumente chamado de *objeto singleton*, que é uma classe que possuirá apenas uma única instância. A declaração acima contrói tanto a classe chamada `HelloWorld` quanto sua intância, também chamada de `HelloWorld`. Esta instância é criada sob demanda, no momento do seu primeiro uso.

O leitor mais astuto pode ter percebido que o código do método `main` não é declarado como `static` aqui. Isso ocorre por que membros estáticos (métodos ou campos) não existem em Scala. Ao invés de usar métodos estáticos, o programador Scala declara esses membros como objetos singleton.

2.1 Compilando o exemplo

Para compilar nosso código você deve usar `scalac`, o compilador Scala. `scalac` funciona como a maioria dos compiladores: ele recebe um fonte como argumento e talvez algumas opções, e produz um ou mais arquivos objeto. Os arquivos objetos aqui produzidos são arquivos .class padrão Java.

Se você salvar o código acima num arquivo chamado `HelloWorld.scala`, você pode o compilar usando o comando abaixo (o sinal de maior ">" representa o prompt de comando e não deve ser digitado):

```
> scalac HelloWorld.scala
```

Isso irá gerar uma série de classes no diretório corrente. Uma dessas classes será chamada de `HelloWorld.class`, e contém a classe que deve ser diretamente executada pelo comando `scala`, como será mostrado a seguir.

2.2 Rodando o exemplo

Uma vez compilado, um programa Scala pode ser facilmente executado através do comando `scala`. Seu uso é bem parecido com o do comando `java`, que é usado para executar programas Java, e aceita as mesmas opções. O exemplo acima pode ser executado com o comando a seguir, que produz a saída esperada:

```
> scala -classpath . HelloWorld

Hello, world!
```

3 Integrando com o Java

Um dos maiores poderes da Scala é sua capacidade de integração fácil com o Java. Todas as classes do pacote `java.lang` são importadas por padrão, enquanto que todas as outras podem ser importadas explicitamente.

Vamos ver um exemplo que demonstra isso. Nós precisamos obter e formatar a data corrente de acordo com a convenção usada num país específico, que no nosso exemplo será a França [1].

A biblioteca de classes Java define um poderoso conjunto de classes utilitárias, como `Date` e `DateFormat`. Como Scala interage diretamente com Java, não existem classes equivalentes na biblioteca de classes Scala: nós simplesmente importamos os pacotes Java correspondentes:

```
import java.util.{Date, Locale}
import java.text.DateFormat
import java.text.DateFormat._

object FrenchDate {
  def main(args: Array[String]) {
    val now = new Date
    val df = getDateInstance(LONG, Locale.FRANCE)
```

[1] Outras regiões como os falantes de francês em parte da Suiça usam a mesma convenção

```
    println(df format now)
  }
}
```

O **import** da Scala é bem parecido com o seu equivalente em Java, mas é ainda mais poderoso. Multiplas classes podem ser importadas de um mesmo pacote através do uso de chaves, como na primeira linha. Outra diferença é que quando desejamos importar todas as classes de um pacote usamos o sublinhado (underscore) (_) ao invés do asterisco (*). Isso ocorre por que o asterisco é um identificador válido em Scala (por exemplo, o nome de um método), como veremos mais a frente.

O **import** na terceira linha traz todos os membros da classe `DateFormat`. Isso faz o método estático `getDateInstance` e o campo estático `LONG` diretamente visiveis.

Dentro do método `main` nós primeiro criamos uma instância da classe `Date` do Java, que por padrão contém a data atual. A seguir, nós definimos o formato da data usando o método estático `getDateInstance`, que importamos anteriormente. Finalmente, nós imprimimos a data atual formatada de acordo com a instância localizada de `DateFormat`. Esta última linha mostra uma propriedade interessante da linguagem Scala: métodos que recebem apenas um argumento podem ser usados com uma sintaxe não fixa. Desta forma, a expressão

```
df format now
```

é apenas outra forma menos extensa de escrever esta expressão

```
df.format(now)
```

Isto pode ser um detalhe menor da sintaxe, mas possui consequências importantes, as quais serão exploradas com mais detalhes na próxima seção.

Para concluir esta seção sobre integração com Java, você deve notar que é possível herdar a partir de classes Java e implementar interfaces Java diretamente em Scala.

4 Tudo é um objeto

Scala é uma linguagem orientada a objetos pura, no sentido de que *tudo* é um objeto, incluindo números e funções. Isso difere de Java, já que Java distingue tipos primitivos (como `Boolean` e `Int`) de tipos de referência, e não permite que sejam manipulados funções como valores.

4.1 Números são objetos

Como números são objetos, eles podem ter métodos. E, de fato, uma expressão aritimética como a abaixo:

```
1 + 2 * 3 / x
```

consiste exclusivamente de chamadas de métodos, por que ela é equivalente a expressão abaixo, como vimos na seção anterior:

```
1.+(2.*(3./(x)))
```

Isso quer dizer que +, *, etc. são identificadores válidos em Scala.

4.2 Funções são objetos

Por mais surpreendente que possa parecer para um programador Java, funções também são objetos em Scala. Isso possibilita passar funções como argumentos e outras funcionalidades interessantes. A possibilidade de manipular funções como valores é um dos mais interessantes paradigmas da chamada *programação funcional*.

Um exemplo muito simples de como isso pode ser útil é o uso de funções como valores, para isso vamos considerar uma função do tipo timer que deve executar uma ação a cada segundo. De que forma passamos a ação para execução? De maneira simples, através de uma função. Esse simples uso de "'passar uma função"' deve ser familiar para muitos programadores: é usado em códigos de interface de usuário, para registrar as funções de callback que devem ser chamadas quando algum evento ocorre.

No exemplo a seguir, a função de timer é nomeada `oncePerSecond`, e recebe uma função de callback como argumento. O tipo desta função é escrito como `() => unit` e este é o tipo de todas as funções que não recebem argumentos e não retornam nada (o tipo `unit` é similar ao `void` do C/C++). A função main deste programa simplesmente chama essa função de timer com uma callback que imprime uma sentença no terminal. Em outras palavras esse programa irá imprimir eternamente a sentença "o tempo corre como um raio" a cada novo segundo:

```
object Timer {
  def oncePerSecond(callback: () => unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def timeFlies() {
    println("o tempo corre como um raio...")
  }
  def main(args: Array[String]) {
    oncePerSecond(timeFlies)
  }
}
```

Note que, para imprimir esta sentença, nós precisamos usar o método `println` sem a necessidade do uso de `System.out`.

4.2.1 Funções anônimas

O código anterior foi simples de entender, e podemos o refinar ainda mais. Primeiro de tudo é necessário entender que a função `timeFlies` é definida apenas para ser passada como parâmetro para a função `oncePerSecond`. Nomear uma função com esta características é desnecessário, sendo mais interessante construir essa função no momento em que a passamos para a `oncePerSecond`. Isso é possível em Scala através do uso do conceito de *funções anônimas*, que são exatamente o que parecem: funções sem nome. Uma versão atualiada de nosso programa de timer que usa uma função anônima no lugar de `timeFlies` pareceria com o abaixo:

```
object TimerAnonymous {
  def oncePerSecond(callback: () => unit) {
    while (true) { callback(); Thread sleep 1000 }
  }
  def main(args: Array[String]) {
    oncePerSecond(() =>
      println("o tempo corre como um raio..."))
  }
}
```

A presença de uma função anônima neste exemplo é revelada pelo símbodo de '=>' que separa os argumentos da função de seu corpo. Neste exemplo, a lista de argumentos está vazia, o que pode ser visto pelo par de parenteses sem nada dentro a esquerda da flecha. O corpo da função é o mesmo que tinhamos na extinta `timeFlies`, do exemplo acima.

5 Classes

Como vimos acima, Scala é uma linguagem orientada a objetos, e dessa forma possui o conceito de classes.[2]Essas classes, em Scala, são declaradas usando uma sintaxe muito parecida com a do Java. Uma das diferenças mais marcantes é que classes em Scala podem ter parâmetros. Isso é ilustrado com a definição abaixo de uma classe para números complexos:

```
class Complex(real: Double, imaginary: Double) {
  def re() = real
  def im() = imaginary
}
```

Essa classe Complex recebe dois argumentos, que são a parte real e a parte imaginária de um número complexo. Estes argumentos devem ser passados no mo-

[2]Para evitar chateações: sabemos que algumas linguagens orientadas a objeto não possuem o conceito de classes, mas Scala não é uma dessas.

mento da criação de uma instância da classe `Complex`, da seguinte maneira:

```
new Complex(1.5, 2.3)
```

A classe contém dois métodos, chamados `re` e `im`, que dão acesso a ambas as partes do número.

Repare que o tipo de retorno desses dois métodos não é especificado explicitamente. Ele será definido automaticamente pelo compilador, que olha os métodos e deduz que o valor de retorno de ambos é um `Double`.

O compilador, porém, pode não estar sempre apto a saber qual o tipo de retorno, e não há uma regra simples para saber qual o tipo que ele efetivamente usou. Na prática isso não é um problema visto que o compilador sabe que só pode mudar o tipo que não foi explicitamente passado. Como uma dica, o programador iniciante em Scala devem tentar omitir tipos quando esses forem fáceis de perceber no contexto, e ver como o compilador se comporta. Após algum tempo o programador terá um bom feeling sobre que tipo pode ou não omitir.

5.1 Métodos sem argumentos

Um pequeno problema dos métodos `re` e `im` é que, para os chamar, deve-se usar um par de parenteses vazios ao lado de seu nome, como pode ser visto abaixo:

```
object ComplexNumbers {
  def main(args: Array[String]) {
    val c = new Complex(1.2, 3.4)
    println("imaginary part: " + c.im())
  }
}
```

Seria interessante acessar a parte real e a parte imaginária como campos, sem a necessidade de colocar esse par de parenteses ao lado do nome. Isso pode ser feito em Scala através da definição de métodos *sem argumentos*. Nossa classe `Complex` pode ser reescrita como abaixo:

```
class Complex(real: Double, imaginary: Double) {
  def re = real
  def im = imaginary
}
```

5.2 Herança e polimorfismo

Todas as classes em Scala herdam de uma super-classe. Quando não é informada de qual super-classe deverá herdar, como no nosso exemplo `Complex`, é usado por padrão `scala.Object`.

Desta forma é possível sobrescrever métodos herdados da super-classe. Em Scala é

obrigatório especificar que o método está sendo sobrescrito através do uso do modificador **override**, para evitar sobrescritas acidentais. Por exemplo, nosso código da classe `Complex` pode ser ampliado com a redefinição do método `toString`, herdado da classe `Object`.

```
class Complex(real: Double, imaginary: Double) {
  def re = real
  def im = imaginary
  override def toString() =
    "" + re + (if (im < 0) "" else "+") + im + "i"
}
```

6 Classes case e localização de padrões

Um tipo de estrutura de dados que costuma aparecer em softwares é a árvore. Por exemplo, interpretadores e compiladores usualmente representam programas internamente como árvores; documentos XML são árvores; e diversos tipos de contêineres são baseados em árvores.

Nós agora iremos examinar como árvores são representadas e manipuladas em Scala através de um programa simples que simula uma calculadora. O objetivo deste programa é manipular expressões aritméticas simples, compostas de somas, constantes do tipo inteiro e variáveis. Dois exemplos dessas expressões são $1+2$ e $(x+x)+(7+y)$.

Nós precisamos primeiro decidir qual a representação que usaremos para as expressões. A mais natural é uma árvores, onde os nós são as operações (no nosso caso, adição) e o restante são os valores (no caso constantes e variáveis).

Em Java, uma árvore pode ser representada usando uma super-classe abstrata, e uma sub-classe concreta por sub-classes concretas. Em linguagens de programação funcionais nós podemos usar um tipo algébrico para o mesmo propósito. Scala possui o conceito de *classes case* que é um meio termo entre ambos. Abaixo você pode ver um exemplo onde definimos a árvore para nossa calculadora:

```
abstract class Tree
case class Sum(l: Tree, r: Tree) extends Tree
case class Var(n: String) extends Tree
case class Const(v: Int) extends Tree
```

Perceba que, pelo fato das classes `Sum`, `Var` e `Const` serem declaradas como classes case, elas diferem de classes padrão em vários aspectos:

- a palavra chave **new** não é necessária para criar instâncias dessas classes (quer dizer que podemos escrever simplesmente `Const(5)` ao invés de **new** `Const(5)`),

- funções do tipo getter são automaticamente definidas para os parâmetros do construtor (por exemplo, é possível pegar o valor do parâmetro `v` de uma instância chamada `c` da classe `Const` apenas com `c.v`),
- a definição padrão dos métodos `equals` e `hashCode` são disponibilizadas, trabalhando com a *structure* das instâncias e não com sua identidade.
- uma definição padrão para o método `toString` também é disponibilizada, e imprime o valor na sua "forma padrão" (por exemplo, a expressão $x+1$ imprime `Sum(Var(x),Const(1))`),
- instâncias dessas classes podem ser decompostas através de *busca por padrão* como veremos mais a frente.

Agora que nós temos definidos os tipos de dados que representam nossa expressão aritímética nós podemos iniciar a definir os operadores para manipulá-las. Nossas expressões irão iniciar com uma função que irá avaliar a expressão em um *ambiente*. O objetivo do ambiente é dar valores as variáveis. Por exemplo, a expressão $x+1$ avaliada em um ambiente que associa o valor 5 a variável x, escreve $\{x \rightarrow 5\}$, dando 6 como resultado.

Dessa forma temos que encontrar um jeito de representar ambientes. Nós podemos, claro, usar algumas estruturas de dados associativas, como uma tabela hash, mas nós podemos usar diretamente funções! Um ambiente nada mais é do que uma função que associad um valor a uma variável. O ambiente $\{x \rightarrow 5\}$ mostrado acima pode ser escrito como abaixo em Scala:

```
{ case "x" => 5 }
```

Esta notação define uma função que, quando recebe a string `x` como um argumento, retorna o inteiro 5, e gera uma exceção se for diferente disso.

Antes de escrever a função de avaliação deixe-me dar um nome ao tipo dos ambientes. Nós podemos, claro, sempre usar o tipo `String => Int` para ambientes, mas simplificaria o programa se nós introduzirmos um nome para este tipo, o que permite modificações simples no futuro. Isso pode ser feito em Scala como mostrado abaixo:

```
type Environment = String => Int
```

A partir de agora o tipo `Environment` (ambiente) pode ser usado como um apelido para funções que vão de um `String` para um `Int`.

Nós podemos agora definir a função de avaliação. Conceitualmente ela é muito simples: o valor de uma soma de duas expressões é simplesmente o valor da soma dessas expressões; o valor da variável é obtido diretamente pelo ambiente e o valor da constante é o valor da constante por sí só. Expressar isso em Scala não é mais difícil:

```
def eval(t: Tree, env: Environment): Int = t match {
  case Sum(l, r) => eval(l, env) + eval(r, env)
  case Var(n)    => env(n)
  case Const(v)  => v
}
```

Essa função de avaliação funciona realizando uma *busca de padrão* na árvore `t`. Intuitivamente o significado de algumas dessas definições pode ficar mais claro:

1. a primeira verificação da árvore `t` é o `Sum`, então é criada uma sub-árvore auxiliar a esquerda numa variável chamada `l` e uma sub-árvores numa variável `r`, e então segue com a avalição da expressão que está após a flecha; esta expressão pode (e faz) uso das variáveis seguidas pelo padrão que aparece antesda flecha, `l` e `r`,
2. se a primeiraq verificação não for bem sucedida então a árvore não é um `Sum`, então é verificado se `t` é um `Var`; se o é então ele anexa o nome contido em `Var` para uma variável `n` e continua com a avaliação na expressão,
3. se a segunda verificação falha então `t` não é nem um `Sum` e nem um `Var`, então ele verifica se é um `Const`, e se o for, então ele anexa o valor contido no nó em `Const` na variável `v` e continua a verificação,
4. finalmente, se todas as verificações falharem, uma exceção é lançada para sinalizar a falha da busca por um padrão; isso aconteceria aqui apenas se mais de uma sub-classe de `Tree` fosse declarada.

Nós verificamos que a idéia básica da busca de padrão é tentar achar um valor numa série de padrões, e quando o padrão é encontrado, extrair e nomear as várias partes do mesmo para, finalmente, avaliar um código que tipicamente faz uso dessas partes nomeadas.

Um programador experiente em orientações a objetos pode ficar supreso por que não definimos `eval` como um *método* da classe `Tree` e suas subclasses. Nós poderíamos fazer isso sem problemas, visto que Scala possibilita a definição de métodos em classes case assim como em classes normais. Decidir usar a busca por padrão ou métodos é uma questão de gosto, mas isso possui implicações importantes em relação a extensibilidade da aplicação:

- quando usamos métodos fica fácil de adicionar um novo tipo de nó, bastando para isto apenas definí-lo em uma sub-classe da `Tree`; Em contrapartida, adicionar uma nova operação para manipular a árvore é uma tarefa tediosa, pois isto requer modificações em todas as subclasses de `Tree`
- quando usamos busca por padrão a situação é invertida: adicionar um novo tipo de nó requer que modificação em todas as funções nas quais a busca por padão atua, para ter o novo nó devidamente; Em contrapartida, adicionar

uma nova operação é fácil, precisando apenar definí-la como uma função independente.

Para explorar bastante buscas de padrões, vamos definir outra operação com expressões aritméticas: derivação simbólica. O leitor deve lembrar as seguintes regras referentes a esta operação:

1. a derivada de uma soma é a soma de suas derivadas,
2. a derivada de qualquer variável v é um se v é a variável relativa na qual a derivação ocorre senão será zero,
3. a derivada de uma constante é zero.

Estas regras podem ser traduzidas quase que literalmente para código Scala, para obter a seguinte definição:

```
def derive(t: Tree, v: String): Tree = t match {
  case Sum(l, r) => Sum(derive(l, v), derive(r, v))
  case Var(n) if (v == n) => Const(1)
  case _ => Const(0)
}
```

Esta função introduz dois novos conceitos relacionados à busca de padrões. Primeiramente, a expressão **case** para as variáveis possui uma *proteção*, uma expressão que segue a palavra-chave **if**. Esta proteção previne que a busca de padrões tenha sucesso a não ser que esta expressão seja verdadeira. Aqui é usado para ter certeza que retornaremos a constante 1 apenas se o nome da variável que está sendo derivada é o mesmo é o mesmo da variável de derivação v. A segunda nova funcionalidade da busca de padrões usada aqui é o *caractere curinga*, escrito como underline, no qual é uma busca de padrões de qualquer valor, sem precisar nomeá-lo.

Nós ainda não exploramos todo o poder da busca de padrões, mas iremos parar por aqui para manter este documento resumido. Ainda queremos ver como as duas funções acima funciona num exemplo real. Para este propósito vamos escrever uma simples função `main`, na qual realiza várias operações sobre a expressão $(x + x) + (7 + x)$: primeiro é computado seu valor no ambiente $\{x \to 5, y \to 7\}$, para então computar sua derivada relativa a x e então y.

```
def main(args: Array[String]) {
  val exp: Tree = Sum(Sum(Var("x"),Var("x")),Sum(Const(7),Var("y")))
  val env: Environment = { case "x" => 5 case "y" => 7 }
  println("Expression: " + exp)
  println("Evaluation with x=5, y=7: " + eval(exp, env))
  println("Derivative relative to x:\n " + derive(exp, "x"))
  println("Derivative relative to y:\n " + derive(exp, "y"))
}
```

Executando esse programa nós temos a saída esperada:

```
Expression: Sum(Sum(Var(x),Var(x)),Sum(Const(7),Var(y)))
Evaluation with x=5, y=7: 24
Derivative relative to x:
 Sum(Sum(Const(1),Const(1)),Sum(Const(0),Const(0)))
Derivative relative to y:
 Sum(Sum(Const(0),Const(0)),Sum(Const(0),Const(1)))
```

Examinando a saída, nós podemos ver que o resultado da derivada deve ser simplificada antes de ser apresentada ao usuário. Definir uma função de simplificação básica, usando busca de padrões, é uma problema interessante(mas surpreendentemente capcioso) e é deixado como um exercício para o leitor.

7 Mixins

Além da herança de código de uma superclasse, uma classe Scala também pode importar código de um ou vários *mixins*.

Talvez a forma mais fácil para um programador java entender o que são mixins é vê-los como interfaces na qual podem também conter código. Em Scala, quando uma classe é sub-classe de mixin, ela implementa aquela interface mixin e herda todo o código contido no nele.

Para ver a utilidade dos mixins, veremos um exemplo clássico: objetos ordenados. Geralmente é útil ser capaz de comparar objetos de uma dada classe através delas mesmas, como por exemplo para ordená-las. Em Java, objetos nos quais são comparáveis implementam a interface `Comparable`. Em Scala nós podemos fazer um pouco melhor do que em Java, definindo nosso equivalente de `Comparable` como um mixin, no qual nós chamaremos de `Ord`

Ao comparar objetos, seis diferentes predicados podem ser úteis: menor, menor ou igual, igual, não igual, maior ou igual e maior. Contudo, definindo todos ele é trabalhoso, especialmente considerando que quatro desses seis podem ser expressos usando os dois remanescentes, isto é, dado os predicados igual e menor (por exemplo), um pode expressar os outros. Em Scala, todas estas observações podem ser facilmente capturadas pela seguinte declaração do mixin:

```
trait Ord {
  def < (that: Any): Boolean
  def <=(that: Any): Boolean = (this < that) || (this == that)
  def > (that: Any): Boolean = !(this <= that)
  def >=(that: Any): Boolean = !(this < that)
}
```

Esta definição cria um novo tipo chamado `Ord`, no qual atua o mesmo papel da interface `Comparable` em Java e padroniza as implementações dos três predicados em termos de um quarto abstrato. Os predicados para igualdade e diferença não aparecem aqui pelo motivo de que são padrões presentes em todos os objetos.

O tipo `Any` que é usado acima, é um super-tipo de todos os outros tipos em Scala. Ele pode ser visto como uma versão mais geral do tipo `Object` em Java, mas também é um super-tipo dos tipos básicos como `Int`, `Float`, etc.

Para fazer objetos de uma classe serem comparáveis, é mais do que suficiente apenas definir os predicados que testam igualdade e inferioridade, e então misturar o código de `Ord` à classe. Como um exemplo, definiremos uma classe `Date`, representando datas no calendário gregoriano. As datas são compostas por um dia, um mês e um ano e são representados como inteiros. Começaremos a definição da classe `Date` como o seguinte:

```
class Date(y: Int, m: Int, d: Int) extends Ord {
  def year = y
  def month = m
  def day = d

  override def toString(): String = year + "-" + month + "-" + day
```

A parte importante aqui é a declaração de **`extends`** `Ord` que é segue do nome da classe e dos parâmetros. Isto declara que a classe `Date` é uma sub-classe da classe `Ord` como mixin.

Nós redefinimos o método `equals` que foi herdado de `Object`, então ele comparará corretamente as datas, comparando seus campos individualmente. A implementação padrão de `equals` não deve ser usada, pois como em Java ele compara os métodos fisicamente. Nós chegamos na seguinte definição:

```
  override def equals(that: Any): Boolean =
    that.isInstanceOf[Date] && {
      val o = that.asInstanceOf[Date]
      o.day == day && o.month == month && o.year == year
    }
```

Este método faz o uso dos métodos já predefinidos `isInstanceOf` e `asInstanceOf`. O primeiro, `isInstanceOf`, corresponde ao operador `instanceof` do Java e retorna verdadeiro se, e apenas se, o objeto no qual foi aplicado, é uma instância do tipo dado. O segundo, `asInstanceOf`, corresponde ao operador de *cast* do Java: se o objeto é uma instância do tipo dado, ele é visto como tal, senão uma exceção `ClassCastException` é lançada.

Finalmente, o último método para definir é o predicado no qual testa a inferioridade, como a seguir. Este faz uso de outro método pré-definido, `error`, que lança uma exceção junto a dada mensagem de erro.

```
  def <(that: Any): Boolean = {
    if (!that.isInstanceOf[Date])
      error("cannot compare " + that + " and a Date")

    val o = that.asInstanceOf[Date]
    (year < o.year) ||
    (year == o.year && (month < o.month ||
                        (month == o.month && day < o.day)))
  }
```

Assim completamos a definição da classe `Date`. Instâncias dessa classe podem ser vistas como datas ou como objetos comparáveis. Além do mais, elas todas definem os seis predicados de comparação mencionados acima: `equals` e `<`, pois eles aparecem diretamente na definição da classe `Date`, e os outros por causa da herança do mixin `Ord`.

Mixins são úteis em outras situações a mais do que as mostradas aqui, com certeza, mas discutir todas as suas aplicações está fora do escopo deste documento.

8 Generalização

A última característica de Scala que exploraremos neste tutorial é a generalização. Programadores Java devem estar bem prevenidos sobre os problemas causados pela falta de generalização em sua linguagem, uma deficiência que é abordada no Java 1.5.

Generalização é a habilidade de escrever código parametrizado por tipos. Como exemplo, um programador escrevendo uma biblioteca para listas encadeadas encontra o problema de decidir qual tipo dar para os elementos da lista. Desde que esta lista foi concebida para ser usada em diferentes contextos, não é possível decidir que o tipo dos elementos terão de ser, por exemplo, `Int`. Isto pode ser totalmente arbitrário e muito restritivo.

Programadores Java contornam isto usando *Object*, que é o super-tipo de todos os objetos. Contudo, esta solução está longe da ideal, desde que não funcionará para os tipos básicos (`Int`, `Long`, `Float`, etc.) e implicando que vários *type casts* terão de ser inseridos pelo programador.

Scala torna possível definir classes (e métodos) genéricos para resolver este problema. Vamos examinar isto com um exemplo do container de classe mais simples possível: uma referência na qual pode ser vazia ou apontar para um objeto de algum tipo.

```
class Reference[a] {
  private var contents: a = _

  def set(value: a) { contents = value }
```

```
  def get: a = contents
}
```

A classe `Reference` é parametrizada por um tipo, chamado `a`, que é o tipo do seu elemento. Este tipo é usado no corpo da classe como o tipo da variável `contents`, do argumento do método `set` e do tipo de retorno do método `get`.

O exemplo do código acima mostra variáveis em Scala que não precisamos explicar mais. Contudo é interessante ver qua o valor dado inicialmente para a variável é `_`, que representa o valor padrão. Este valor padrão é 0 para os tipos numéricos, **`false`** para o tipo booleano, `()` para o tipo `unit` e **`null`** para todos os tipos de objetos.

Para usar esta classe `Reference`, alguem precisa especificar qual o tipo para se usar no parâmetro de tipo a no qual é o tipo contido pela célula. Como exemplo, ao criar e usar uma célula guardando um inteiro, alguém deve escrever assim:

```
object IntegerReference {
  def main(args: Array[String]) {
    val cell = new Reference[Int]
    cell.set(13)
    println("Reference contains the half of " + (cell.get * 2))
  }
}
```

Como pode ser visto no exemplo, não é necessário fazer *cast* no valor retornado pelo método `get` antes de usá-lo como um inteiro. Também não será possível guardar nada além de um inteiro naquela célula particularmente, pois foi declarada para guardar um inteiro.

9 Conclusão

Este documento apresenta uma rápida visão geral da linguagem Scala com alguns poucos exemplos. Caso deseje aprofundar-se mais recomendamos que leia o *Scala By Example*, que possui muito mais exemplos, e consulte a *Scala Language Specification* quando necessário.

10 Sobre a tradução

Este documento foi traduzido por Marcelo Castellani e Thiago Rocha, membros da lista de discussão Scala-Br. Para participar visite a página a seguir:

http://groups.google.com/group/scala-br