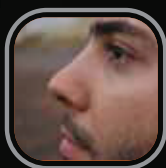


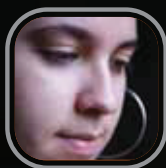
# Conhecendo as Principais Linguagens para JVM

Saiba quais são e as principais características das principais linguagens disponíveis para a JVM



**Felipe Rodrigues de Almeida**


(felipero@gmail.com), é engenheiro de Software com 8 anos na área de TI. Ao longo da carreira trabalhou em projetos críticos para empresas como Novell, IBM, Telefônica, CPqD e Embraer. Recentemente se especializou em software de colaboração on-line e redes sociais, atuando na Fratech. Agilidade e expressividade tem sido o seu dia-a-dia que varia entre mentoring técnico e coaching de equipes ágeis. É um dos criadores da Academia do Agile para a Fratech e Globalcode.



**Flavia Castro de Oliveira**

(flavia@fratech.net), é desenvolvedora de software com 2 anos de experiência. Tem trabalhado principalmente com linguagens dinâmicas em projetos de desenvolvimento pela Fratech. Gosta de design, bom html, javascript e css. É usuária de Mac.

*A cada dia, mais e mais linguagens aparecem no mercado e a tendência é que mais de uma linguagem seja utilizada na mesma aplicação. Isso cria a necessidade de nos tornarmos desenvolvedores políglotas. A plataforma Java tem acompanhado esta tendência através de implementações customizadas de diferentes linguagens, todas prontas para serem executadas em integração com código Java. Neste artigo veremos as principais linguagens dessa plataforma, focando em suas implementações para a JVM, dando destaque a suas principais características e funcionalidades.*



Desde o surgimento dos primeiros computadores, a maior preocupação dos cientistas da computação sempre foi encontrar um jeito simples e fácil para passarmos as instruções necessárias para a máquina e assim obtermos os resultados de seu processamento. Isso acontece porque, ao contrário do que os leigos podem pensar, computadores não têm inteligência, apenas poder de processamento. Por isso, cabe a nós, desenvolvedores de software, determinar qual é o resultado esperado para um determinado processamento. Isso mesmo! Os computadores não são capazes de criar soluções, eles apenas seguem instruções determinadas por seus programadores.

Com isso em mente, desde o início da computação buscamos evoluir a forma como passamos instruções às máquinas. Isso fez com que surgissem linguagens de programação com diferentes estilos e sintaxes. Cada estilo está normalmente atrelado a um ou mais paradigmas de programação. Após algum tempo, por criatividade ou por estratégia de mercado, empresas e pessoas criaram linguagens derivadas de outras linguagens, que misturavam paradigmas e modificavam sintaxe com o intuito de atender a um nicho específico da computação, como paralelismo, segurança ou mesmo programação concorrente. Essa evolução de linguagens é fundamental para a evolução das tecnologias, pois as necessidades mudam com o passar do tempo. Com isso em mente, é fácil determinar que seria impossível criar uma única linguagem que atenda a todos os gostos e critérios, por isso, não vale a pena iniciar discussões sobre qual linguagem é melhor ou pior.

Neste texto iremos fazer um tour pelas principais linguagens disponíveis para a JVM, analisando seus pontos fortes e aprendendo um pouco sobre sua sintaxe e integração com a plataforma Java. Falaremos sobre as linguagens Scala, Ruby, Clojure, Python, JavaScript e Groovy. Cada uma com uma característica e um atrativo diferente. Você terá a oportunidade de comparar as linguagens através dos exemplos e optar pela próxima linguagem que deseja aprender.

## Linguagens Dinâmicas vs Linguagens Estáticas

Muito tem se falado em relação a linguagens dinâmicas e linguagens estáticas mas qual é a diferença real?

A diferença está na forma que a linguagem trata os tipos das variáveis de um programa. Nas linguagens de tipagem estática, as variáveis são como recipientes que só aceitam um único tipo, normalmente especificado pelo programador ou inferido pelo compilador. Ao tentar atribuir um dado de tipo diferente a uma variável, o compilador irá identificar e emitir um erro, reclamando que os tipos não são compatíveis. Em linguagens dinâmicas o compilador não se preocupa com o tipo das variáveis de dos dados envolvidos, logo não há mensagens dizendo que o tipo é incompatível. Há vantagens e desvantagens nos dois casos. Linguagens estáticas diminuem a chance de erros do programador, pois há uma etapa a mais de verificação no processo. Essa etapa de verificação não existe nas linguagens dinâmicas, permitindo que erros possam acontecer em tempo de execução. Por isso, em linguagens dinâmicas o uso de testes automatizados é ainda mais importante. Mas então, por que a aderência às linguagens dinâmicas cresce cada vez mais? A resposta para essa pergunta está nos paradigmas de metaprogramação. A tipagem estática se baseia no fato de que se um objeto é de um determinado tipo, irá sempre se comportar como um objeto daquele tipo. A metaprogramação permite que os objetos (e seus comportamentos) sejam modificados em tempo de execução, seja através de adição ou modificação de métodos e variáveis. Nesse caso um objeto pode ter o mesmo tipo de outro, mas possuir características únicas que o diferencia. Isso implica em um poder que permite criar funcionalidades e ter uma flexibilidade nas linguagens dinâmicas que não é possível alcançar em linguagens estáticas. Podemos utilizar conceitos de AOP para simular metaprogramação em linguagens de tipagem estática, mas a flexibilidade não é a mesma e normalmente o esforço é muito maior.

## Linguagens para a JVM

A máquina virtual do Java (Java Virtual Machine ou JVM) é um software que abstrai grande parte da complexidade de passar instruções para a máquina. Isso permite que utilizemos linguagens mais amigáveis aos humanos para dar instruções claras a um computador. A JVM foi lançada em meados de 1995 em sua primeira versão e ela dava suporte à execução de uma linguagem binária chamada bytecode. Na época Java era a única linguagem que possuía um compilador que gerava código compilado em bytecode. Com a evolução e estabilização da linguagem Java e da virtual machine juntamente com algum uso de JNI (Java Native Interface), é possível executar outras linguagens de programação utilizando a plataforma Java, a partir da JVM. Muitas dessas linguagens são totalmente escritas em Java, sendo chamadas de linguagens pure Java. Outras são apenas implementações de compiladores que transformam o código-fonte em código bytecode, executável na JVM. O intuito de executar uma linguagem na JVM é tirar proveito de uma plataforma estável e evoluída, com

aceitação nos mais diversos ambientes de desenvolvimento e grande comunidade. O ecossistema Java é gigantesco e responsável por grande parte das aplicações corporativas presentes na atualidade. Dessa forma é possível tirar proveito de todo esse ecossistema e ainda assim, utilizar a linguagem que melhor se adequar à sua equipe e projeto. Atualmente, mais de 300 linguagens podem ser executadas na JVM, incluindo algumas das linguagens mais cobiçadas no momento.

## O projeto Da Vinci

Em meados de 2006 surgiu uma iniciativa de suportar linguagens externas a serem invocadas a partir de código Java na JVM. Com isso, surgiram algumas implementações ainda modestas de linguagens aderentes a este modelo. Essa iniciativa resultou na JSR-223 – Scripting for the Java Platform. A JSR-223 já facilitava a integração mais alinhada de outras linguagens com a JVM, mas ainda deixava a desejar devido a restrições presentes no compilador Javac. Isso porque segundo a JSR-223, o código-fonte das linguagens seria carregado como texto (String) a partir de programas Java e então interpretado através de uma chamada de um método do interpretador específico daquela linguagem. Isso caracterizaria todas as linguagens como linguagens de script, por causa deste modelo de interpretação. Linguagens de script são interessantes para várias coisas, mas não devem ser usadas sem algum outro programa que as interprete. No início de 2007 um grupo de desenvolvedores iniciou um projeto inicialmente chamado de MLVM (Multi-Language Virtual Machine). Este projeto culminou em uma nova JSR, a JSR-292 – Supporting Dynamically Typed Languages on the Java Platform. Este projeto também é conhecido como “the Da Vinci Machine Project” e tem o objetivo de estender a JVM para permitir uma integração ainda maior com outras linguagens, através de suporte built-in a alguns paradigmas específicos. Com este projeto, poderemos evoluir ainda mais no suporte a outras linguagens sem a necessidade de contornar a diferença entre paradigmas. A intenção é que o JDK 7 já venha com essas extensões.

## Scala

Scala é uma linguagem interessante que mistura alguns paradigmas e pode ser executada na JVM. Seu desenvolvimento iniciou em 2001 por Martin Odersky como um estudo de junção entre o paradigma funcional e de objetos e atualmente está na versão 2.8. Possui uma comunidade ativa e envolvente, composta por grandes desenvolvedores. A linguagem tem uma personalidade mais rebuscada em termos de sintaxe e funcionalidades. Ela utiliza conceitos dificilmente utilizados quando se está programando em Java. Isso torna a curva de aprendizado um pouco mais demorada, mas nem por isso menos prazerosa. Scala é utilizada como linguagem de propósito geral, sem um foco específico, pois possui grandes recursos para utilização em código de backend e aplicações web, através do framework Lift que apresenta uma abordagem alternativa de padrão para a web. Além disso, possui grandes características em relação à programação concorrente, que junto com sua integração com o Java é ótima para aplicações desktop com Swing. Enfim, não há restrições quanto ao uso dessa linguagem, e a comunidade tem se aproveitado disso.

Scala mistura dois grandes paradigmas de programação: programação funcional e orientada a objetos. Isso exige do desenvolvedor maturidade para discernir quando um ou outro paradigma deve ser utilizado. Em outras palavras, pode-se alcançar o melhor dos dois mundos com Scala. Do aspecto de orientação a objetos, tudo em Scala é um objeto, o que remete à pura OO do Smalltalk. Possui Classes e Traits (um conceito parecido com o de classes abstratas e

interfaces). Suporta herança única e grande flexibilidade através de composição baseada em mixins. Os mixins surgem como uma ótima alternativa à herança múltipla e, em Scala, são alcançados através de Traits. Para atender ao paradigma de programação funcional, Scala define que todas as funções são valores (portanto, objetos). Oferece suporte a funções anônimas (lambda) e a high-order functions, ou seja, funções que recebem e/ou retornam outras funções, fator de extrema importância para a programação funcional. Oferece funções aninhadas e currying de funções, que significa redefinir funções com valores padrão para alguns parâmetros da função original. A sua sintaxe é de certa forma inspirada na sintaxe do Java, porém com várias diferenças que podem surpreender. É uma linguagem de tipagem estática e seu sistema de tipagens suporta Classes Genéricas, Anotações de Variância, Inner Classes, Tipos Abstratos, Composição, Métodos Polimórficos, dentre outras coisas. Além disso o sistema de tipagem de Scala conta com um poderoso mecanismo de inferência de tipos. Ele pode identificar o tipo de um objeto mesmo que o programador não anote especificamente o tipo desejado. Esses recursos tornam Scala uma das linguagens mais impressionantes do momento. Scala conta ainda com um sistema de concorrência muito eficiente, baseado no conceito de Actors oriundo da linguagem erlang. Possui mecanismos impressionantes de Pattern Matching para controle de fluxo e operações com Expressões Regulares.

## Instalação e execução de código Scala

A instalação é tão simples quanto descompactar um arquivo e adicionar seus binários ao PATH de sua máquina. Faça o download do pacote para sua plataforma no link <http://www.scala-lang.org/downloads>. Descompacte o arquivo em um diretório de sua preferência e defina as variáveis de ambiente conforme a figura 1.

Environment	Variable	Value (example)
Unix	\$SCALA_HOME	/usr/local/share/scala
	\$PATH	\$PATH:\$SCALA_HOME/bin
Windows	%SCALA_HOME%	c:\Progra-1\Scala
	%PATH%	%PATH%;%SCALA_HOME%\bin

Figura 1. Variáveis de Ambiente de Scala.

### Executando código Scala

Para executar código Scala há várias opções. Scala é uma linguagem compilada, portanto, possui um compilador que pode ser acionado através do comando `scalac`. O compilador irá gerar um arquivo `.class` contendo bytecode para a JVM. A partir daí podemos executar esse `.class` utilizando o comando `scala`:

```
scala NomeDaClasse
```

Ou utilizando o próprio comando `java`:

```
java NomeDaClasse
```

O detalhe a observar é que quando utilizamos o comando `java`, é necessário adicionar a biblioteca `scala-library.jar` ao classpath.

Por poder ser tratada como uma linguagem de script, há ainda a possi-

bilidade de executar código Scala sem compilar (na verdade a compilação ocorre, porém de forma transparente). Para isso, basta salvar seu código em um arquivo com a extensão `.scala` e utilizar o comando `scala` para executá-lo, como no exemplo abaixo:

```
scala arquivo.scala
```

Scala ainda oferece um shell iterativo onde expressões scala são interpretadas imediatamente. Para acessar este shell execute o comando `scala` sem passar nenhum argumento. Digite `:help` no shell do Scala para informações sobre como utilizar o shell. Veja na figura 2 o console Scala em ação.

```
[master] % cd /data/workspaces/examples/scala_intro % scala
Welcome to Scala version 2.7.7.final (Java HotSpot(TM) Server VM, Java 1.6.0_16).
Type in expressions to have them evaluated.
Type :help for more information.

scala> class Teste {
    |   val t = "um atributo"
    | }
defined class Teste

scala> val t = new Teste()
t: Teste = Teste@12345678

scala> val t = new Teste
t: Teste = Teste@12345678

scala> t.t
res0: java.lang.String = um atributo

scala> println(t)
um atributo

scala>
```

Figura 2. Exemplo de um console Scala.

Para aqueles que gostam de utilizar o maven, há ainda um plugin para scala que suporta a compilação e a execução de código Scala integrado ao ciclo de vida de projetos maven.

### Exemplos

Vejamos agora o código de Hello World do Scala na Listagem 1. Este código ilustra a estrutura de um código mais básico em Scala.

#### Listagem 1. Hello World em Scala.

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

Scala oferece ainda mecanismos para programação concorrente muito poderosos como o Actors. Actors são objetos que podem receber mensagens e enviar mensagens, e são executados em sua própria Thread. Podem ser síncronos ou assíncronos. Na Listagem 2 vemos um exemplo do uso de Actors em Scala.

Nesse exemplo, criamos um actor e o atribuímos ao value `meuActor`. Um Actor possui um corpo, similar a um método, que será executado em uma thread separada. Podemos então enviar mensagens a um actor. Este actor poderá receber as mensagens através do método `receive`, que recebe um bloco de código para tratar as mensagens, podendo utilizar o mecanismo de pattern matching do Scala para definir como tratar cada tipo de mensagem recebida. Um actor também pode enviar mensagens a outro, através do método `!`, como em `meuActor ! "hello actors"`.



Listagem 2. Actor em Scala.

```
import scala.actors.Actor._

val mainThread = self

val meuActor = actor {
  println("Número de mensagens até agora? " + mailboxSize)
  mainThread ! "enviado" // Envia a mensagem
  Thread.sleep(3000) // Processa alguma coisa
  println("Número de mensagens enquanto eu estava ocupado? " + mailboxSize)

  receive {
    case msg =>
      println("Mensagem recebida " + msg)
  }

  mainThread ! "recebido"
}

receive { case _ => }

println("Enviando mensagem ")

meuActor ! "hello actors"

receive { case _ => }
```

Analisando o código da Listagem 2, ao criamos o actor ele inicia seu processamento em uma Thread separada, porém fica na fila aguardando sua vez de entrar em execução. Enquanto isso, a thread principal continua sua execução e os resultados podem ser variados neste caso. O meuActor começa sua execução e envia uma mensagem à mainThread, após imprimir uma frase no console. Após enviar a mensagem ele é posto pra “dormir” através do método sleep do Java. Nesse momento, a mainThread volta a execução. Utilizamos o método receive para receber a mensagem do meuActor, mas apenas a ignoramos.

Em seguida enviamos uma mensagem ao meuActor e posteriormente deixamos um receive esperando pra tratar outra mensagem qualquer. Após os 3 mil milissegundos, o meuActor volta à fila para ser executado novamente. Quando isso acontece, ele imprime o número de mensagens em seu mailbox através da variável mailboxSize que registra o número de mensagens que um actor recebeu. Executa então o método receive onde apenas imprime a mensagem que recebeu e após o término, envia uma mensagem de volta à mainThread. Este exemplo mostra um pouco do poder de Scala para programação concorrente.

## JRuby

Criado em 2001 por Jane Arne Petersen, o JRuby é uma implementação da linguagem de programação Ruby para rodar sobre a plataforma da JVM. O Jruby está na versão 1.5.1 que é compatível com o ruby 1.8.7 mas contém vários dos recursos do ruby 1.9.x, tornando-o uma boa opção para se manter atualizado em relação a novidades do ruby. A linguagem Ruby oferece aos desenvolvedores um conjunto de funcionalidades e características poderosas que permitem código mais expressivo e elegante que a maioria das linguagens. O JRuby estende essas capacidades para trabalhar dentro da plataforma Java. O código

Ruby é eficiente, bonito e fácil de ler. Em Ruby, números e strings são objetos, ou melhor tudo é objeto. Isso significa que usando Ruby você economiza tempo e seu código fica mais limpo. É muito utilizado por aqueles que desejam criar aplicações utilizando Rails e outras APIs do mundo ruby, mas que não querem abrir mão da JVM. Seu ponto forte é a criação de aplicações para a web, aproveitando os conceitos de convenção ao invés de configuração presentes em sua linguagem. Com JRuby é possível, por exemplo, executar Rails no Google App Engine.

Executar código JRuby é muito fácil. O JRuby vem com um executável chamado jruby. Esse executável é suficiente para executar qualquer arquivo que contenha código ruby utilizando a seguinte sintaxe: jruby <<ARQUIVO>>.rb. Assim como o Ruby, o JRuby conta com um shell interativo, o irb, que pode ser usado para testar código ruby instantaneamente. Podemos acessar o irb digitando irb no terminal. Para finalizar, ainda é possível utilizar o jrubyc, uma ferramenta que irá compilar o código jruby em bytecode java, resultando em um arquivo .class. Uma vez que você possui um arquivo .class é possível executá-lo utilizando o comando java, mas não esqueça de colocar os jars do jruby em seu classpath. Uma das principais vantagens do Ruby é sua comunidade. Ela é composta por pessoas interessadas em evoluir a plataforma e escrever código de qualidade. Isso fez com que o ecossistema do ruby crescesse muito, oferecendo diversas opções para testes unitários e funcionais, bibliotecas para diversas tarefas do dia-a-dia e mecanismos

Listagem 3. Exemplo de mixim em JRuby.

```
module Montadora
  def release_model(model, year = 2010)
    if @models[year] then @models[year] << model else @
      models[year] = [model] end
    end
  end

  class GM
    include Montadora

    attr_accessor :models

    def initialize()
      @models = {2007 => ['Celta', 'Astra', 'Corsa'], 2008 => ['Celta',
        'Astra', 'Corsa']}
    end
  end

  gm = GM.new

  puts gm.models

  gm.release_models('Vectra GT')
  gm.release_models('Vectra GT', 2009)
  gm.release_models('Astra')

  puts gm.models
```

de reúso para evitar retrabalho. Ruby é uma linguagem de tipagem dinâmica que oferece poderosos recursos de metaprogramação. Possui suporte a funções anônimas (lambda) e a closures. Uma das principais características da linguagem é seu “açúcar sintático”. Algumas tarefas são muito mais fáceis de serem executados em ruby do que em outras linguagens. Vejamos um exemplo disso na Listagem 3. Veja como criamos um module chamado Montadora. Neste exemplo um module

funciona como um repositório de métodos. Eles podem ser incluídos uma classe, que passará a possuir os mesmos métodos do module. Fazemos isso através do include Montadora. Esse é um mecanismo de mixin muito utilizado no Ruby para modularização e reuso.

## Integração com Java

O JRuby se integra perfeitamente com Java, sem grandes problemas e permite que você trabalhe com as APIs do Java em seu código JRuby como se fossem bibliotecas do Ruby. Isto lhe proporciona o benefício de reutilizar as bibliotecas disponíveis para a plataforma Java.

Listagem 4. Chamando Java a partir do JRuby.

```
include Java

include_class "java.util.TreeSet"

set = TreeSet.new
set.add "foo"
set.add "Bar"
set.add "baz"
set.each { |v| puts "value: #{v}" }
```

Na Listagem 4 vemos como podemos importar classes Java através do método include\_class. include\_class é um método disponível no module Java incluso na primeira linha do programa na Listagem 4. Importamos a classe TreeSet. Logo abaixo criamos uma instância desta classe como se fosse uma classe ruby, através do método new. A partir daí, podemos utilizar os métodos e ler os atributos como qualquer outro objeto ruby.

### Instalação

A maneira mais fácil de rodar JRuby é fazendo o download do arquivo binário mais recente no site do JRuby em <http://jruby.org/download>. Faça o download conforme a sua plataforma, extraia os arquivos em um local de sua preferência e adicione o diretório para sua variável de ambiente. Mais detalhes podem ser encontrados nessa página: <http://jruby.org/getting-started>

## Clojure

Clojure é um dialeto LISP que surgiu em meados de 2007 a partir de um esforço de Rich Hickey, que continua à frente do projeto, agora na versão 1.1.0. Pode ser executado na JVM e possui grande interoperabilidade com Java. O código é compilado para bytecode Java. Por ser um dialeto LISP, segue a filosofia de code-as-data (ou seja, o próprio código é considerado um dado), além da sintaxe característica do LISP que muitas vezes assusta alguns desenvolvedores. Clojure é uma linguagem funcional que não possui conceitos de orientação a objetos, mas que oferece estruturas de dados ricas e interessantes. Por sua característica funcional, Clojure é utilizada no backend de aplicações que exigem muito processamento e performance, pois possui excelente suporte a programação concorrente e lazy-evaluation, que permite melhor aproveitamento dos recursos físicos, facilitando inclusive a clusterização de sua execução. O foco da linguagem é em programação concorrente e multi-threaded, possuindo características importantes para isso. As estruturas de dados são imutáveis por padrão. Isso evita as famosas

“race conditions” resultando em dados síncronos através das threads. O fato de utilizar o paradigma funcional, através de high-order functions ajuda. Outro fator importante é a aderência de lazy-sequences, característica de dialetos LISP, que reduz drasticamente a quantidade de dados persistidos em memória. Clojure também possui estruturas de dados mutáveis, mas estas devem ser utilizadas em conjunto com o STM, Software Transactional Memory, que trata alterações nos conjuntos de dados como transações atômicas, garantindo mais uma vez a integridade dos dados. O grande desafio de Clojure é integrar o mundo totalmente funcional do LISP com o mundo de orientação a objetos do Java. No entanto, Clojure consegue achar formas de realizar essa integração e, após uma curva de aprendizado, é possível tirar grande proveito de seu poder e ao mesmo tempo usufruir das bibliotecas existentes para Java. Em relação à sua sintaxe, Clojure, como qualquer outro dialeto LISP, possui uma sintaxe muito simples e composta por pouquíssimos elementos. Programas escritos em Clojure normalmente serão menores que em outras linguagens. Como um dialeto de LISP, Clojure traz vários dos benefícios do LISP, mas também alguns problemas. O principal deles é a obsessão do LISP por parênteses. Do ponto de vista de orientação a objetos, Clojure traz o conceito de Structs, estruturas de dados com atributos, para armazenamento de dados de forma similar a um hash. Os objetos Java são tratados como structs, mas isso é transparente para o desenvolvedor. Podemos criar objetos em Java e utilizá-los dentro de Clojure.

### Exemplos

Na Listagem 5 vemos um código Clojure que exemplifica a criação de uma função e de um struct.

A sintaxe, apesar de incomum, é bem simples. Para executar uma função devemos envolvê-la em parênteses e listar seus argumentos separados por espaços. Podemos passar outras funções como argumentos. No exemplo da Listagem 5 criamos um Struct chamado Car e depois iniciamos a criação de várias instâncias diferentes de Car. Neste código, tudo que não é uma função é um argumento. Até funções podem ser argumentos. Para criar um struct, utilizamos a função defstruct que recebe o nome do struct e seus atributos como argumentos. Para criar uma instância de um struct, utilizamos a função struct que recebe no struct do qual você quer uma instância e os valores para os atributos do struct. Para definir uma variável, utilizamos a função def que recebe o identificador da variável e seu valor. Na Listagem 5, ao definirmos a variável palio, passamos como valor o resultado da execução de uma função struct, ou seja, a nova instância do struct Car é atribuída à variável. Para acessar um atributo dessa nova instância, tratamos o atributo como uma função (pois é uma função), executando (:atributo palio).

Ainda na Listagem 5, na declaração da variável golf, utilizamos algo mais sofisticado. Uma função anônima. Criamos o struct passando “Golf” para o atributo :model e (fn [] (-2013 3)) como parâmetro para o atributo :year. Essa função anônima não possui parâmetros (note os colchetes vazios []) e seu corpo executa uma outra função, a função – (subtração). Para terminar, temos um exemplo de definição de função na Listagem 5 e o uso desta função sendo executada e seu valor retornado na definição do struct atribuído à variável celta. Essa função chamada whichModel recebe um único parâmetro branch. Os parâmetros das funções são declaradas dentro de colchetes. Para executar essa função, basta envolvê-la em parênteses e listar seus argumentos separados por espaço. Vemos ainda um exemplo de uso da função if.

## Listagem 5. Funções e Structs em Clojure

```
(defstruct Car :model :year)
(def palio (struct Car "Palio" 2007))

(println palio)

; Acessando um atributo
(:model palio)

; No lugar do atributo :year passamos uma função anônima
(def golf (struct Car "Golf" (fn [] (- 2013 3))))

; Executamos a função anônima obtendo o valor dela e envolvendo-a em
parênteses
((:year golf))

; Declaração de uma função chamada whichModel que recebe um único
argumento
(defn whichModel
  [brand]
  (if (= brand "GM")
    ("Celta")
    ("Palio")))

; Criamos um struct que utiliza a função whichModel para descobrir o
modelo do carro
(def celta (struct Car (whichModel "GM") 2007))
(println (str "Modelo -> " (:model celta)))
```

## Listagem 6. Fibonacci em clojure, usando lazy sequences

```
; lazy-seq-fibo
(defn lazy-seq-fibo
  ([])
  (concat [0 1] (lazy-seq-fibo 0 1)))
([a b]
  (let [n (+ a b)]
    (lazy-seq
     (cons n (lazy-seq-fibo b n)))))

; Best fibo ever
(defn fibo []
  (map first (iterate (fn [[a b]] [b (+ a b)]) [0 1])))
```

A Listagem 6 nos apresenta muitas outras funcionalidades presentes em Clojure. Uma delas é o conceito de lazy-sequences. Sequences é uma lista lógica de dados. Abstrata, pois não segue uma implementação específica de estrutura de dados, apenas caracteriza uma interface comum. Lazy sequences são sequences que possuem uma fórmula para calcular seu valor e permanecem sem ser processadas, para não consumir memória. Quando precisamos obter um determinado valor de uma lazy sequence, utilizamos alguma função, como first ou nth para obter o valor de uma determinada posição. Nesse momento a sequence irá processar o mínimo necessário para obter o valor desejado. Lazy sequences são muito úteis para casos em que o conjunto é infinito. No exemplo da Listagem 6 mostramos duas funções que fazem a mesma coisa – buscam um determinado número fibonacci. A primeira função, chamada lazy-seq-fibo foi construída manualmente, usando recursos de mais baixo nível. A função chamada apenas de fibo já utiliza uma estrutura de map, proveniente do conceito de map-reduce e um ite-

rator que já são lazy por natureza. Por seguir o paradigma funcional é comum que Clojure use e abuse da recursividade como pode ser visto na primeira função da Listagem 6. A função chama a si própria várias vezes. Nenhum estado é armazenado em variáveis que não sejam argumentos da função. Isso facilita o aproveitamento dos processadores e minimiza o uso de memória. Clojure também permite total aproveitamento de classes Java. Na Listagem 7 vemos um exemplo disso.

## Listagem 7. Acessando Java a partir de Clojure.

```
; Utilização de classes java
(def rnd (new java.util.Random))
(println (. rnd nextInt))

; Importação de classes
(import '(java.util Random)
        '(java.text MessageFormat))

; Construindo novas instâncias
(def rnd (new Random))
; Construindo novas instâncias de forma mais curta
(def rnd (Random.))

; Chamada de métodos com parâmetros
(println (. rnd nextInt 200))

; Métodos e atributos estáticos
(println (. Math PI))
(println (System/currentTimeMillis))
```

A forma para acessar classes Java está bem fácil de entender na Listagem 7. Há o caminho mais longo e o caminho mais curto que utiliza atalhos sintáticos. Podemos importar as classe que estão em nosso classpath. Conceitos como polimorfismo e herança são ignorados, pois não utilizamos classes, apenas instâncias dos objetos, lembrando que os tipos dos objetos também são ignorados pelo interpretador de Clojure. Para simplificar, todos os conceitos de OO podem ser tratados em objetos Java que são utilizados normalmente dentro de Clojure.

Apesar de ser uma linguagem nova e que adota um paradigma diferente, Clojure tem sido utilizada no backend de diversas webapps, incluindo do site <http://flightcaster.com/>. Há muitos outros casos de sucesso, e sua comunidade tem se mostrado muito ativa.

## Instalação

Para instalar Clojure basta você fazer o download da versão mais recente do pacote nesse link: <http://code.google.com/p/clojure/downloads/list>, descompactar o arquivo no diretório de sua preferência e definir as variáveis de ambiente.

## Python

O Python é uma linguagem dinâmica com muitos recursos agradáveis ao programador e com maior flexibilidade. Criada no final dos anos 80, é uma das linguagens mais utilizadas para desenvolvimento gráfico em ambientes \*unix. Boa parte do projeto KDE, incluindo plugins e add-ons são escritos em Python. O google também aposta forte nessa linguagem. O Google App Engine foi

inicialmente disponibilizado para aplicações escritas em Python.

Como linguagem, o Python oferece recursos poderosos de metaprogramação e que possibilitam a aderência ao paradigma funcional. O resultado é que um determinado trecho de código pode ser expressado de diversas formas. Muitos de seus recursos se assemelham ao Ruby, e sua comunidade é tão intensa quanto a do Ruby ou do Java. No final de 1997, Jim Hugunin criou o Jython, uma implementação do Python executada na JVM, com o intuito de substituir C por Java em casos de código que exigiam performance intensiva dentro de programas Python. O jython está atualmente na versão 2.5.1. Qualquer código Java pode ser chamado através do Jython. Além disso, os principais módulos da linguagem Python estão disponíveis para uso imediato. Uma das principais vantagens de produtividade do código Jython é a tipagem dinâmica. Você não precisa declarar variáveis e nem precisa dizer ao interpretador qual o tipo de dados esperado de uma variável. Essa abordagem é similar à do Ruby e do JavaScript.

Além da tipagem dinâmica, Jython tem um interpretador de linha de comando onde você pode digitar qualquer expressão Jython e executá-las de forma interativa, usufruindo assim de um ambiente rápido e simples de aprendizagem e debug.

Outra característica muito interessante sobre Python é a questão da indentação do código. Os blocos de código são delimitados por espaços, não existem símbolos de “abre” e “fecha”, como, por exemplo, (begin e end) ou { chaves } como no Java. No lugar desses itens, o Python requer uma tabulação padronizada de espaços. A indentação é obrigatória. Sendo assim é recomendado usar uma IDE de Python que faz a indentação automática, pois em um editor de texto comum, torna-se necessário indentar manualmente, possibilitando diversos erros. A Listagem 8 mostra o certo e o errado na indentação em python.

Listagem 8. Identação em Python.

```
# Identação correta
def valor1(self):
    try:
        self.c = input('First Value: ')
        c = self.c
        return c
    except:
        print 'Invalid!'
        self.valor1()

# Identação incorreta
def valor1(self):
try:
    self.c = input('First Value: ')
    c = self.c
    return c
except:
    print 'Invalid!'
    self.valor1()
```

A maior parte da biblioteca padrão do Python está inclusa no Jython. Embora alguma das funcionalidades sejam iguais as da biblioteca padrão do Java, muitas não são. Nesse aspecto, podemos citar o suporte a expressão regular, parsers, suporte a rede,

frameworks de testes unitário etc. A integração entre Python e Java através do Jython é muito direta e simples. É permitido aos desenvolvedores misturar livremente as duas linguagens durante o desenvolvimento e distribuição de suas aplicações. Pacotes Java podem ser importados para o Jython como se fossem Jython modules e os objetos podem ser criados usando a sintaxe de criação de objetos do Jython.

## Metaprogramação em Python

Listagem 9. Metaclass em Python.

```
class Montadora(type):
    def __new__(cls, name, bases, attrs):
        print "Inicializando a classe", name

        if 'models' in attrs:
            for model in attrs['models']:
                attrs[model] = model.upper()
            return type.__new__(cls, name, bases, attrs)

class GM:
    __metaclass__ = Montadora
    models = ["celta", "corsa", "astra", "vectra"]

print "Modelos:", GM.models
print GM.celta
print GM.corsa
print GM.astra
print GM.vectra
```

Na Listagem 9 vemos um exemplo de metaprogramação em Python. Todas as classes em Python (e Jython) possuem uma metaclass. A metaclass é a classe da classe, o que a torna consequentemente uma classe também. Por padrão, a metaclass das classes é a classe type. Na Listagem 9 nós criamos nossa própria metaclass para adicionarmos atributos de forma dinâmica. Para isso, criamos a metaclass Montadora que estende type. Essa classe sobrescreve o método `__new__`, que é chamado no momento do carregamento de uma classe, ou seja, quando utilizamos uma classe pela primeira vez. O método `__new__` recebe quatro parâmetros: a classe que está sendo inicializada; o nome dessa classe como uma string; as superclasses dessa classe; e um dictionary (HashMap) com o atributos dessa classe. No método `__new__` da nossa classe Montadora, definimos que, se houver um atributo no nosso dictionary de atributos com o nome “models”, então, será assumido que models contém uma lista de strings e para cada valor desta lista, adicionamos um atributo na classe, cujo valor é o nome do atributo convertido para maiúsculo. Para usufruir da nossa metaclass, ao declarar a classe GM, ainda na Listagem 9, declaramos que o valor do atributo `__metaclass__` é Montadora. Logo abaixo, declaramos o atributo models que recebe uma lista de strings, como esperamos no método `__new__`. Agora vem a parte esclarecedora. Devido à nossa “magia negra”, a classe GM possui o atributo models, que declaramos, mas também possui os atributos celta, corsa, astra e vectra. Podemos então obter os valores desses atributos executando `GM.celta`, `GM.astra`, `GM.corsa`, `GM.vectra`, como mostrado na Listagem 9.

## Instalação

A instalação do Jython é uma das mais fáceis, pois possui um wizard. Basta realizar o download de um jar executável através do link <http://www.jython.org/downloads.html> e executá-lo utilizando o comando:



## java -jar jython\_installer-2.5.1.jar

Siga o wizard escolhendo cuidadosamente entre as opções oferecidas. Ao final, coloque o binário do Jython em seu path, para que você tenha mais conforto ao utilizá-lo.

## Java Script com Rhino

JavaScript é uma das linguagens mais utilizadas em todo o mundo. Isso aconteceu por uma série de fatores, mas principalmente porque é a linguagem padrão para interação com web browsers. Ser uma das linguagens padrão, junto com o HTML e o CSS, foi o que impulsionou a popularidade da linguagem, mas ao mesmo tempo, minimizou o poder e as características dela, pois grande parte do código JavaScript disponível no mundo é cópia feita por web designers que nem imaginam o que é um objeto. JavaScript é uma linguagem extremamente poderosa, com recursos interessantes que merecem respeito Criada por volta de 1995, cresceu muito rápido por ser utilizada com foco em websites. Com o surgimento da web 2.0 iniciou-se uma busca por interatividade e usabilidade que resultou em algo chamado Ajax. A onda então é criar sites e aplicações que usufruem ao máximo do Asynchronous JavaScript and XML. Isso trouxe o JavaScript para a frente dos holofotes e então iniciou-se uma busca por mais qualidade nos códigos do famoso JS. Surgiram então frameworks poderosos e impressionantes, como o JQuery, Prototype, Dojo Toolkit, ExtJS etc. Todos realizando uso de funcionalidades de metaprogramação e abusando do dinamismo e flexibilidade da linguagem. Esse frameworks mostram o poder do JavaScript e nos levam a pensar: “Onde mais, além da web, eu poderia utilizar tais funcionalidades?” O que poucos sabem no entanto é que é possível utilizar JavaScript como uma linguagem de propósito geral, bastando para isso escolher dentre um dos vários JavaScript engines disponíveis no mercado. Temos o V8 do Google, o SpiderMonkey da Mozilla e o mais alguns. Para a JVM, temos uma implementação de JavaScript chamada Rhino. Criado em 1997 como um projeto da Netscape e atualmente mantido pela fundação Mozilla, o Rhino converte JavaScript em classes Java, oferecendo total integração entre o ecossistema Java e o ecossistema JavaScript. Atualmente o Rhino está na versão 1.7. Obviamente os objetos e funções disponibilizadas pelo browser não fazem parte da API do JavaScript, portanto é preciso cuidado ao escrever seus programas fora do browser. O JavaScript é uma linguagem Orientada a Objetos, porém com uma diferença: seu paradigma é a Orientação a Objetos baseada em protótipos. A ideia é que podemos criar objetos que “imitam”, ou baseiam-se em objetos já existentes. Podemos então modificar esse objetos da forma que quisermos, dada a flexibilidade de metaprogramação do JavaScript. Em JavaScript temos os tipos primitivos que são Number (Integer e Float), String e Boolean. Possui também dois tipos triviais, null e undefined. Qualquer outra coisa além disso é um objeto. JavaScript não é uma linguagem que segue o paradigma de programação funcional, porém possui o conceito de funções – blocos de código executáveis que recebem argumento e retornam valores. As funções em JavaScript também são objetos, e portanto podem ser atribuídas a variáveis e passadas como argumentos para outras funções. Quando atribuímos uma função a um atributo de um objeto, dizemos que esta função é um método do objeto.

## Diferenças da OO

Uma das diferenças mais marcantes entre o estilo de OO do JavaScript em relação ao Java é que JavaScript não possui o

conceito de classes. Não é possível criar classes em JavaScript. Há no entanto a possibilidade de simular comportamentos similares a classes, mas na verdade é apenas uma simulação (apesar de que o suporte a classes reais está previsto para a versão 2.0 do JavaScript). Veja um exemplo na Listagem 10.

### Listagem 10. Prototypes em JavaScript.

```
function Person( arg) {
    this.name = arg;
}

var bob = new Person( "Bob");
var alice = new Person( "Alice");
print(bob.name)
```

Repare que no corpo da função declarada, definimos um atributo ao objeto this, que neste caso representa a própria função. O valor atribuído a this.name é o valor do argumento. Logo após a declaração da função, criamos mais duas instâncias, através da palavra new. Isso se parece muito com o processo de criação de uma classe em Java, mas não se engane. Em JavaScript, todos os objetos possuem um atributo chamado prototype, que representa o objeto em si. Este atributo funciona como um Hash com chave:valor, onde cada chave é o nome de um atributo e o valor, bem... é seu valor. Para definirmos um método, basta adicionarmos uma chave e como valor definimos um objeto executável, ou uma função. A palavra new cria uma cópia deste objeto e a atribui às variáveis bob e alice. Note que bob e alice não possuem a função em si, apenas uma cópia do prototype da função, o mesmo objeto representado por this dentro do corpo da função. Logo, print(bob.name) irá imprimir o texto “Bob”. Assim, simulamos o comportamento de uma classe. Analise o código da Listagem 11 para mais um exemplo de como podemos simular classes em JavaScript.

### Listagem 11. Simulando classes em JavaScript.

```
var Car = function() {
    return this;
}

var apolo = Car();
var verona = Car();
var golf = new Car();

apolo == verona // TRUE
apolo === verona // TRUE
apolo == golf // FALSE
apolo === golf // FALSE
```

Os operadores == comparam se o valor existente em uma variável é igual ao valor existente na outra variável. O operador === compara se a referência dentro das variáveis aponta para a mesma instância de objeto. Quando apenas executamos a função Car() obtemos seu valor através do return this presente no corpo da função. Esta função irá retornar a referência do mesmo objeto todas as vezes que for executada. No entanto quando criamos a variável golf, utilizamos a palavra reservada new, chamando uma espécie de construtor e atribuindo à golf, uma cópia do prototype de Car. Apesar disso, não gosto de tratar do prototype com uma forma de simular classes, mas sim como um paradigma diferente de orientação a objetos, baseado em um protocolo de metadados completamente flexível e diferenciado. Podemos

adicionar métodos e atributos aos objetos JavaScript em tempo de execução de forma fácil e flexível. Imagine quantas coisas podem ser realizadas com tal poder e flexibilidade.

## Integração com Java

Qualquer classe Java é acessível a partir do JavaScript no Rhino. O Rhino disponibiliza um objeto global chamado `Packages` que irá encapsular todos os pacotes Java. Dessa forma, basta executar `Packages.<<seupacote>>.<<subpacote>>.<<SUA_CLASSE>>` para ter acesso a uma classe. Para acessar a classe `java.io.File`, por exemplo, podemos nos referir a ela como `Packages.java.io.File`. O Rhino oferece também uma forma de importar classes e pacotes, evitando assim a necessidade de o desenvolvedor ter que referenciar as classes por seu nome completo todas as vezes. Para importar o pacote `java.io` podemos utilizar a função `importPackage` como em `importPackage(java.io)`. Mais detalhes em [https://developer.mozilla.org/en/Scripting\\_Java](https://developer.mozilla.org/en/Scripting_Java). Veja alguns exemplos na Listagem 12.

Listagem 12. Integração com Java.

```
// Acessando classes e constantes
print(Packages.java.lang.Math.PI)

// Importação de classes
importClass(Packages.java.util.Random);

// Instanciação de objetos
var rnd = new Random()

// Chamada de métodos
rnd.nextInt()
rnd.nextInt(200)
```

## Instalação

O Rhino está disponível para todas as plataformas. No caso do Linux, as principais distribuições possuem pacotes prontos para sua instalação. É possível porém baixar os binários do site da Mozilla em [https://developer.mozilla.org/en/Rhino\\_downloads\\_archive](https://developer.mozilla.org/en/Rhino_downloads_archive). O Rhino possui uma classe chamada `Shell` que é responsável por executar código JavaScript e também iniciar um shell no estilo REPL. Para acessá-la, é preciso ter o arquivo `js.jar` em seu `classpath`. Esse arquivo acompanha a distribuição do Rhino. Utilize então o comando `java org.mozilla.javascript.tools.shell.Main`. Mais detalhes nesta página: <https://developer.mozilla.org/en/Rhino/Shell>.

## Groovy

Groovy é uma linguagem de programação orientada a objetos desenvolvida especificamente para a plataforma Java criada por volta de 2003 por James Strachan. Atualmente está na versão 1.7.2 e é utilizada como linguagem de propósito geral, servindo de porto seguro para aqueles que adoram Java, mas que querem uma linguagem mais flexível e dinâmica. Sua integração com Java é nativa e possui sintaxe muito parecida com a sintaxe do Java, muito parecida mesmo. Tão parecida que basta renomear um arquivo `.java` para `.groovy` e utilizá-lo a partir do `groovy`. Isso torna a curva de aprendizado muito curta para aqueles que conhecem o Java. O ecossistema do Groovy oferece ainda algumas bibliotecas adicionais, como os `builder`, que fazem uso das características

dinâmicas do Groovy. Groovy ainda possui características de Python, Ruby e Smalltalk e é uma das linguagens mais estáveis e robustas para JVM. Groovy é uma linguagem de tipagem dinâmica, mas que pode ser utilizada com tipagem estática também. Em outras palavras, a tipagem dinâmica é opcional. Podemos escolher entre especificar o tipo de uma variável ou deixá-la sem tipo específico. Groovy possui suporte a closures, com um sintaxe muito simples de utilizar.

## Exemplos

Listagem 13. Classe groovy.

```
class Car {
    def miles = 0
    String model
    final year

    Car(y) {
        year = y
    }
}

def car = new Car(2008)

println "Year: ${car.year}"
println "Miles: ${car.miles}"
println "Model: ${car.model}"
println 'Setting miles'
car.miles = 25
println "Miles: ${car.miles}"
```

Na Listagem 13 declaramos a classe `Car`. Esta classe possui três atributos, `miles`, `model` e `year`. O atributo `year` é `final`. Significa que, após sua inicialização, não poderá ter seu valor alterado. Para o atributo `miles`, especificamos o tipo, usufruindo assim da verificação de tipos. Quando não queremos especificar o tipo de uma variável, utilizamos a palavra reservada `def`, como na declaração do atributo `miles` do exemplo da Listagem 13.

Na Listagem 14 vemos a definição da classe `PizzaDSL`, que representa um interpretador de uma DSL (Domain Specific Language) em Groovy. Utilizamos recursos de metaprogramação, um dos pontos fortes do Groovy. Nesse caso, o método `methodMissing` é responsável por tratar todas as chamadas a métodos não existentes de um objeto, dessa forma, quando chamamos algum método não existente nesse script, o método `methodMissing` irá apenas adicionar os argumentos ao `HashMap` pedidos. O método `imprimir` é responsável por imprimir os pedidos no shell. Logo após a declaração da classe `PizzaDSL`, utilizamos a classe `java.util.File` para ler um arquivo com código da DSL, o arquivo `orderPizza.dsl` apresentado na Listagem 15. Dividimos e tratamos suas linhas através do método `split()` e do método `trim()`, ambos vindos do `java.lang.String`. Depois, criamos um texto que representa um script groovy. Nesse script, instanciamos um objeto do tipo `PizzaDSL` e o armazenamos na variável `pd`. Para cada linha do nosso arquivo de DSL, mostrado na Listagem 15, adicionamos ao nosso script uma chamada ao método `pd.imprimir()`. Ao final temos uma string que representa nosso script de execução da DSL. Esse script executa o método `imprimir` para cada linha do arquivo `orderPizza.dsl`. Há ainda um truque no método `imprimir`. Ele executa a closure que `lhe` é passada. Uma closure é um bloco de código, como se fosse o corpo de um método, porém sem sua declaração. Cada linha do arquivo `orderPizza.dsl` será

passada como uma closure para o método imprimir que o irá executar. Isso quer dizer que linhas como tamanho grande ou ingredientes Azeitonas, Cebola, Atum serão executadas como código groovy, como se fossem chamadas a métodos. Como esses métodos não existem, o método `methodMissing` será chamado no lugar deles, adicionando o nome do método como chave do `HashMap` pedidos e os argumentos dos métodos como valores para essa chave do `HashMap`.

Listagem 14. DSL em groovy.

```
PizzaDSL {
  def grande = 'large'
  def pequena = 'pequena'
  def fina = 'fina'
  def visa = 'Visa'
  def master = 'Mastercard'
  def Azeitonas = 'Azeitonas'
  def Cebola = 'Cebola'
  def Atum = 'Atum'

  def pedidos = [:]

  def methodMissing(String methodName, args) {
    pedidos[methodName] = args
  }

  def imprimir(index, closure) {
    closure.delegate = this
    closure()
    println "Validando e processando a ordem $index:"
    println pedidos
    pedidos.each { key, value ->
      println "${key} -> ${value.join(', ')}"
    }
    println "\n"
  }
}

def dsl = new File("orderPizza.dsl").text

def orders = dsl.split("\n\n").collect { it.trim() }

def script = """
  def pd = new PizzaDSL()
  """
orders.eachWithIndex { order, index ->
  script += """\n pd.imprimir(${index + 1}) { $order } \n"""
}

new GroovyShell(PizzaDSL.getClassLoader()).evaluate(script)
```

## Instalação

Baixe as distribuições em: <http://groovy.codehaus.org/Download> e <http://grails.org/Download>, descompacte os arquivos, descompacte, configure as variáveis de ambiente, e atualize seu PATH. O único pré-requisito para a instalação do groovy é o Java SDK 1.5 ou superior instalado em seu computador. Lembre-se de que a variável de ambiente JAVA\_HOME deve estar configurada de acordo com sua instalação Java.

## Considerações finais

Como pudemos ver neste artigo, há muitas linguagens de programação que podem executar na JVM. Linguagens para todos os gostos e credos. Cada linguagem com seus pontos fortes e seus pontos fracos. Cada lin-

guagem possui uma filosofia, definida por sua comunidade. O nível de documentação e material disponível também varia conforme a comunidade. O objetivo deste artigo não foi ensinar a utilizar as linguagens, mas oferecer uma visão clara das características de cada uma delas. Vimos quais as principais características das principais linguagens disponíveis para a JVM, incluindo Scala, JRuby, Clojure, Jython, JavaScript (rhino) e Groovy. Cada uma focada nas suas principais forças e funcionalidades mais características. Há ainda outras linguagens disponíveis para JVM num total de mais de 300 implementações diferentes, com destaque para JavaFX, e implementações de PHP. Acompanhe mais artigos nas próximas edições da revista MundoJ. Ao longo de minha carreira, tenho visto diversos desenvolvedores discutindo quais linguagens são melhores ou piores que as outras e isso não resulta em bons resultados. Ao contrário, tenho visto que, cada vez mais, é melhor evoluirmos em direção a aplicações que utilizam mais de uma única linguagem, usando os recursos que cada uma tem de melhor. Precisamos nos tornar desenvolvedores políglotas! Para aqueles que utilizam a plataforma Java, a boa notícia é que ser desenvolvedor políglota está cada vez mais fácil!•

Listagem 15. Código de utilização da DSL

```
tamanho grande
massa fina
ingredientes Azeitonas, Cebola, Atum
endereço "Na casa do Morais/Tartaruga"
pagamento visa, '1234-1234-1234-1234'

tamanho grande
massa fina
ingredientes Azeitonas, Cebola, Atum
endereço "Meio da praia de Itapuã"
pagamento visa, '1234-1234-1234-1234'

tamanho grande
massa fina
ingredientes Azeitonas, Cebola, Atum
endereço "No bar do Neusão"
pagamento visa, '1234-1234-1234-1234'
```



## Para Saber Mais

Veja o artigo "Scala para o MundoJ" da edição 40 da revista MundoJ para mais detalhes sobre a linguagem Scala.

Veja os artigos da edição 39 da revista MundoJ para conhecer o framework Grails e para ver como utilizar JRuby e Cucumber para testes.

Veja o artigo "JRuby da web ao Desktop" da edição 36 da revista MundoJ para mais detalhes sobre JRuby.



## Referências

Sites:

Groovy – <http://groovy.codehaus.org>  
 Jython – <http://www.jython.org/>  
 JRuby – <http://jruby.org/>  
 Rhino – <http://www.mozilla.org/rhino/>  
 Scala – <http://www.scala-lang.org/>  
 Clojure – <http://clojure.org/>