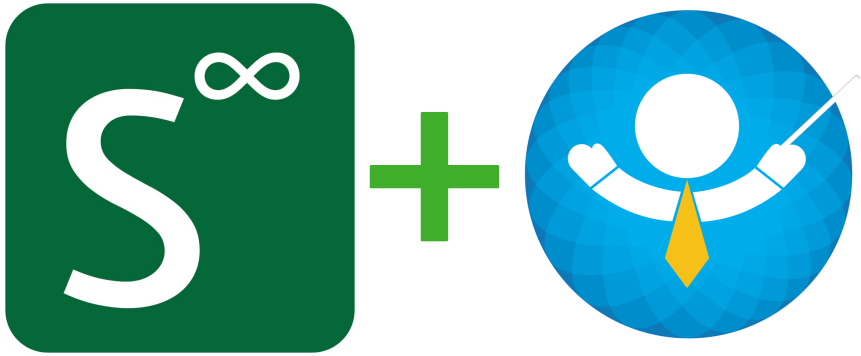
Spark 2 For Devs

Rodolfo Dias



Quem sou eu?

- Rodolfo Dias
  - Sr Data Engineer | Sr Software Engineer @ ScalaSystems
  - Para saber mais visite:
    - linkedin.com/in/rodolfo-duarte-dias-15a05713

Scala Systems

Visão geral de conteúdo

- Spark no mundo de BigData
- Componentes e Arquitetura
- Instalação
- Computação distribuída com RDD's (Resilient Distributed Dataset)
- API do Spark para interação programática com cluster
- Spark SQL para datasets estruturados
- Streaming de dados com Spark Streaming
- Boas práticas e dicas de preparação e instalação de ambiente

Ementa

- Introdução à Scala
- Introdução ao Spark
- Arquitetura e RDD
- Spark SQL, DataSets e DataFrames
- Processamento de dados e performance
- Incrementar performance
- Criação de Aplicações
- Streaming de dados

Lab 0.1: Acessando o Ambiente

Scala Systems

Parte 1: Entendendo o Scala

- Introdução
- Coleções
- Métodos e funções
- Tipos de Objetos e polimorfismo

Parte 1.1: Introdução ao Scala

O que é Scala?

- Uma linguagem de programação.
- Suporta os modelos de **orientação a objetos** e **programação funcional**.
- Compartilha do mesmo runtime do **Java (JVM).**
  - Consequentemente consegue utilizar as bibliotecas do java.

Matei's respondeu a alguns anos atrás porque escolheu Scala:

" When we started Spark, we wanted it to have a concise API for users, which Scala did well. At the same time, we wanted it to be fast (to work on large datasets), so many scripting languages didn't fit the bill. Scala can be quite fast because it's statically typed and it compiles in a known way to the JVM. Finally, running on the JVM also let us call into other Java-based big data systems, such as Cassandra, HDFS and HBase. Since we started, we've also added APIs in Java (which became much nicer with Java 8) and Python "

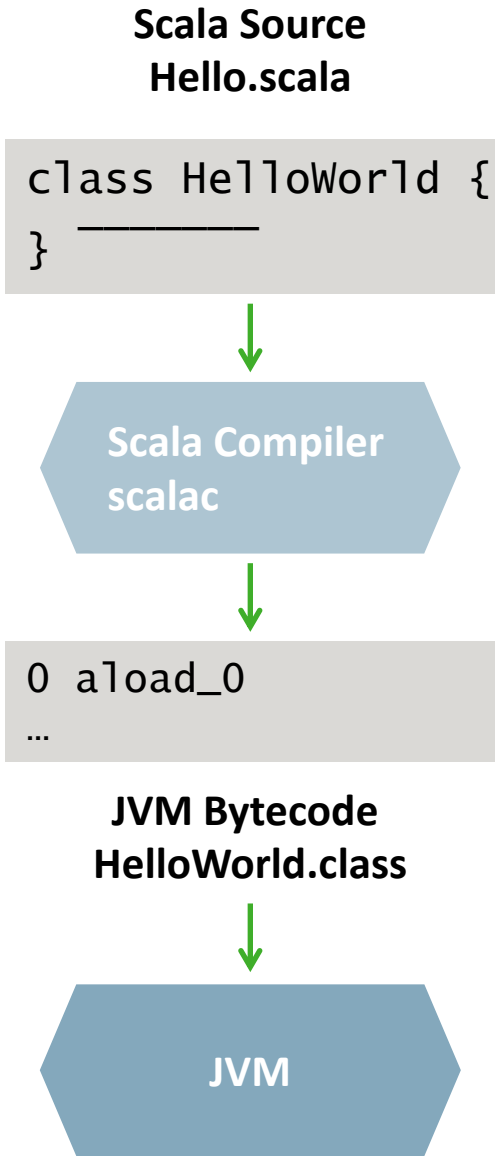
Entendendo o por quê de Scala para Spark

- Habilidade para trabalhar com o **ecossistema Hadoop**, onde sua execução é baseada na JVM.
- Buscar uma linguagem que permita uma **Interface Interativa** e habilidade de suportar a **Programação Funcional** de forma sucinta.

Componentes de execução

- Os códigos escritos em Scala são compilados para ByteCode para serem interpretados e executados pela **JVM.**

- Scala REPL (Read–eval–print loop)
  - Interpretador interativo do Scala.
  - Permite realizar operações singulares.



Definição de variáveis

- **`var`**: Iniciar uma variável dinâmica (mutável)
- **`val`**: Iniciar uma variável constante (imutável)
- Explicitar o tipo da variável é opcional.

```
var x = 4           // Definição de uma variável mutável
x = 7;              // Alteração do valor da variável.

val y = 9           // Definição de uma variável constante
y=11                // error: reassignment to val

var d: Double = 8 // Definição de uma variável com o Tipo explicito

// Erro de incompatibilidade do valor da variável com o Tipo que foi
escolhido.
val z : Int = 1.1
```

Interpretador Scala

- Execute o interpretador utilizando o comando **scala** ou **bin/spark-shell** para o Spark.

```
~# scala
Welcome to Scala <Scala Version> (Java Distribution and version).
Type in expressions for evaluation. Or try :help.


scala> :help
```

Interagindo com o interpretador

- Todos os commandos escritos no terminal serão compilados e interpretados e seus resultados serão retornados.

```
~# scala
Welcome to Scala <Scala Version> (Java Distribution and version).
Type in expressions for evaluation. Or try :help.

scala> var a = 0
a: Int = 0

scala> a + 1
res0: Int = 1

scala> b = a + 1
       ^
       error: not found: value b.    // A variável não foi definida
```

Tipos numéricos e booleanos

- Numéricos: `Byte, Char, Short, Int, Long, Float, Double, Boolean`
  - Tipos numéricos são objetos com métodos de apoio.

- Tipos especiais
  - **`Any`**/**`AnyVal`**: Tipo genérico.
  - **`Unit`**: Tipo que representa um retorno inexistente

Mas por que criar um tipo que representa um retorno?

Operadores

- Pode-se realizar operações aritméticas utilizando os sinais comuns de linguagens de programação. Por exemplo: **`a + b`**
  - Isso é traduzido para **`a.+(b)`** que representa um método que pode ser sobrescrito ou alterado.

- Novos métodos de operadores podem ser adicionados em objetos.

```
> val bi : BigInt = 1
bi: BigInt = 1

> bi + 2  // O mesmo que bi.+(2)
res14: scala.math.BigInt = 3
```

Loops

- Segue o mesmo modelo de operadores **`for`** e **`while`** como C, Java e etc.

```
> val b = 1 to 2
b: scala.collection.immutable.Range.Inclusive = Range(1, 2)

> for (i <- 0 to b.length-1) println(b(i))
1
2

> for (cur <- b) println(cur)
1
2

> var i = b.length-1
> while (i >= 0) {
        println(b(i))
        i -= 1 }
2
1
```

Condicionais

- Possue o mesmo padrão de instruções **`if/else.`**
- As instruções podem retornar valores.

```
> val i = 1
> val j = 2

> if (i>j) {
      | println ("i é maior")
      | } else {
      | println("j é maior")
      | }
j é maior

> val k = if (i>j) i else j    // Armazenando o valor da instrução
k: Int = 2
```

Instrução “match”

- Permite definir alternativas para que possam ser selecionadas baseadas no valor de uma entrada.
  - `case matchValue => expression`

```
> val monthNum = 2

> monthNum match {
      |   case 1 => println(”Janeiro")
      |   case 2 => println(”Fevereiro")
      |   case 3 => println(”Março")
      |   case _ => println(”Mês inválido") // _ = Default case
      | }
February

> val monthName = monthNum match {
      |    case 1 => ”Janeiro"
      |    case 2 => ”Fevereiro"
      |    case 3 => ”Março"
      |    case _ => " Mês inválido "
      | }
monthName: String = February
```

LAB

Lab 0.2: Scala Introdução

Mini-Lab — 5 minutos

- Inicie o REPL do Scala
  - Execute o commando `:help`
  - Veja o comando `:paste` (auxilia na ação de colar um trecho de código)
- Declare variáveis
  - Mutáveis
  - Imutáveis
  - Com tipo explicito
- Utilize Loops e Condicionais

Parte 1.2: Coleções

Classes de coleções

- Classes de coleções
  - **`Seq`**: Sequencias — coleção ordenada(**`List, Vector`**)
  - **`Set`**: Coleção não ordenada, porém sem dados duplicados
  - **`Map`**: Coleção de pares de chave/valor

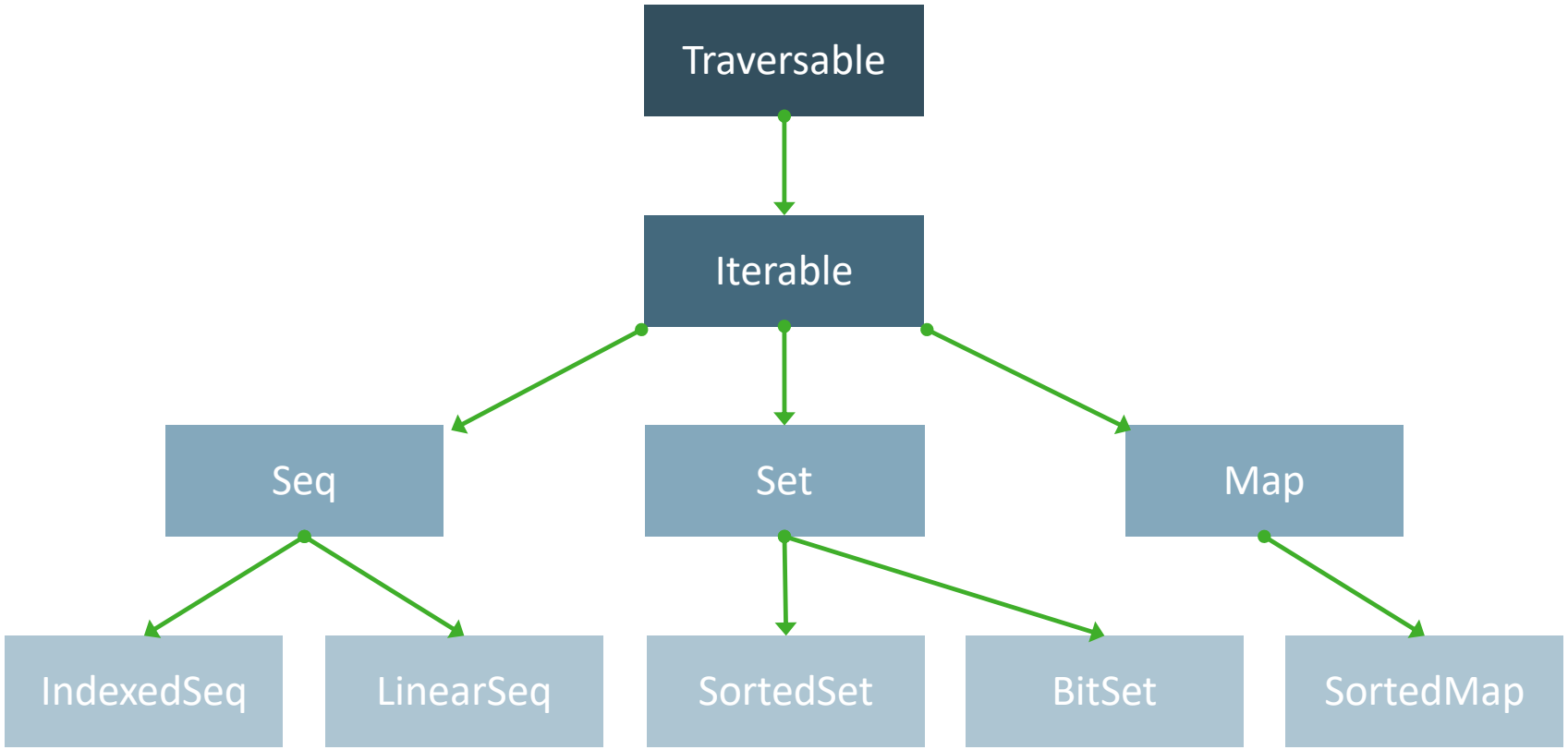
- A criação de Coleções pode ser parametrizada por um Tipo que representa os dados que estão nela.

```
// Criando uma coleção com tipo explicito
> val colors = List[String]("red", "green", "blue")
colors: List[String] = List(red, green, blue)

// Criando uma coleção com o tipo implicito
> val colors = List("red", "green", "blue")
colors: List[String] = List(red, green, blue)
```

Principais tipos de Coleções

- No pacote `scala.collection`



Scala Systems

Criando Instâncias de coleções

- Crie uma instância selecionando o tipo de coleção que você quer
  - Os construtores recebem os dados das coleções
  - Coleções mutáveis (`scala.collection.mutable`) e imutáveis (`scala.collection.immutable`).
  - **Imutáveis são o padrão.**

```
> val comida = Set(”brasileira", ”japonesa", ”arabe”, “brasileira”)  //
Elementos duplicados são unificados
comida: scala.collection.immutable.Set[String] = Set(brasileira, japonesa,
arabe)

// Criando um Mapa chave/valor
> val pratosDoDia = Map(”brasileira"->2, ”japonesa"->10, ”arabe"->3)
pratosDoDia: scala.collection.immutable.Map[String,Int] = Map(brasileira ->
2,
japonesa -> 10, arabe -> 3)

// Criando um “Set” mutável
> val comida = scala.collection.mutable.Set(“brasileira", ”japonesa",
”brasileira")
comida: scala.collection.mutable.Set[String] = Set(brasileira, japonesa)
```

Interagindo com coleções

- As possibilidades de interações mudam em cada classe de Coleção
  - Todas as classes suportam **iteração**
  - **Sequences** (`scala.collection.Seq`) suporta buscar um elemento pelo seu índice
  - **Maps** suportam buscas pela chave

```
> val comida = Set(”brasileira", ”japonesa", ”arabe")
pets: scala.collection.immutable.Set[String] = Set(brasileira, japonesa,
arabe)
> comida.tail
res7: scala.collection.immutable.Set[String] = Set(japonesa, arabe)

> val a = 1 to 5
a: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5)
> a(0)
res10: Int = 1

> val pratosDoDia = Map("brasileira"->2, "japonesa"->1, "arabe"->0)
pratosDoDia: scala.collection.immutable.Map[String,Int] = Map(brasileira ->
2,
japonesa -> 1, arabe -> 0)
> pratosDoDia("brasileira")
res11: Int = 2
```

Métodos de classes de coleções

- Existem muitos métodos para os diferentes tipos de classes de coleções, como por exemplo:
  - **Operações com Maps**(`map, flatMap`): Executar uma função em cada elemento da coleção para criar um novo Map.
  - **Amostragem e filtragem**(`take, filter, slice`): Executa funções que permitem extrair parte do dado, seja filtrado ou em pedaços.
  - **Reductions**(`fold, reduce`): Aplicar uma operação massiva nos elementos, criando um único resultado.

Pares e Tuplas

- Criam um objeto com valores combinados
  - Os valores podem ser de tipos diferentes
  - Para acessa-los, usamos o método `._n` (n representa o índice do valor)

```
> val pair = (”banana", 4)
pair: (String, Int) = (banana,4)

> pair._1
res149: String = banana

> println(pair._2)
4

> val wordCounts = Array((”banana",3), (”pera",2), (”uva",1))
wordCounts: Array[(String, Int)] = Array((banana,3), (pera,2), (uva,1))

> wordCounts(0)._1
res152: String = banana

> val aresta = (1L,2L, 7)
aresta: (Long, Long, Int) = (1,2,7)
```

Parte 1.3: Métodos e funções

Criação de funções

- Basicamente, assim:

```
def function-name(arg1: Type …) : ReturnType =
                          { function-body }
```

  - **Para os parâmetros devemos sempre explicitar o tipo do dado a ser passado**
  - Você pode deixar de lado o uso do retorno e das chaves (1)

```
> def max(x: Int, y: Int): Int = {
        if (x > y) x
        else y
      }
max: (x: Int, y: Int)Int

> def max2(x: Int, y: Int) = if (x > y) x else y
max: (x: Int, y: Int)Int

>  max (1,2)
res79: Int = 2
```

Função anônima

- É possível criar funções sem nomes com um escopo específico
  - Geralmente são usadas para processamento de coleções em massa, como métodos “reduce”

- Basicamente, são definidas assim:
  - `(argumentList) => { functionBody }`

```
> val b = 1 to 2
b: scala.collection.immutable.Range.Inclusive = Range(1, 2)

> b.foreach( (x:Int) => { println(x*x) } )  // Função anônima
1
4

> b.foreach( x => println(x*x) )
```

Função literal

- Usada para armazenar uma função dentro do escopo de variável, geralmente para serem usados em escopos de métodos de coleções.

```
// Declarando apenas o tipo do parâmetro
> val f = (i: Int) => { i%2==0 }
> val f = (_:Int)%2==0
f: Int => Boolean = <function1>

// Declarando o tipo do parâmetro e o tipo do retorno
> val f: (Int) => Boolean = i => { i % 2 == 0 }
> val f : Int => Boolean = i => i%2==0
> val f : Int => Boolean = _%2==0
```

Usando os métodos filter e map

- **`filter`**: Selecionar os elementos baseando-se em um função de filtragem.

- **`map`**: Cria uma nova coleção aplicando funções em cada elemento da coleção anterior.

```
> val b = 1 to 4
b: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4)

// Filtrando valores pares
> b.filter(a => a%2==0)
 > b.filter(f) // Filtrando valores pares utilizando a função literal
               // do slide anterior
res2: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4)

// Utilizando o map para criar uma nova coleção com os quadrados dos
// elementos
> b.map(a => a*a)
res3: scala.collection.immutable.IndexedSeq[Int] = Vector(1, 4, 9, 16)
```

Função de múltiplos argumentos e reduce

- Funções comuns, anônimas e literais podem possuir mais de um único argumento, como temos no seguinte exemplo.

```
> val sum = (a:Int, b:Int) => a+b
sum: (Int, Int) => Int = <function2>

> (1 to 3).reduce(sum)
res21: Int = 6

// Função anônima
> (1 to 3).reduce( (a,b) => a+b )
res22: Int = 6
```

O operador _ (Underscore)

- _ (underscore) podem ser usados como parâmetros de funções.

```
> (1 to 4).filter( a => a%2 == 0)
res30: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4)

> (1 to 4).filter(_%2 == 0)  // Operador _ representando um parâmetro
res31: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4)

> (1 to 3).reduce( (a:Int, b:Int) => a+b ) // Função anônima
res22: Int = 6

> (1 to 3).reduce( _+_ )   // Operador _ representando dois parâmetros
res113: Int = 6
```

LAB

Lab 0.3 com Coleções e Funções

Mini-Lab — 10 minutos

- Use o interpretador (scala REPL)
- Crie instâncias de diferentes classes e utilize os métodos e funções
  - Range de valores utilizando “`to`”
  - `filter`
  - `map`
  - `reduce`

Parte 1.4: Tipos de Objetos e polimorfismo

Criando uma classe

- Classes são blueprints de objetos.
- Exemplo de criação de classe:

```
> class Square (val size : Int) {
     |      def area : Int = size * size
     |      def perimeter : Int = size*4
     |      def display : Unit = {
     |         println("Size = " + size)
     |         println("Area = " + area)
     |     }
     | }
defined class Square

> val sqr = new Square(6)
sqr: Square = $iwC$$iwC$Square@478eb50

> sqr.area
res145: Int = 36

> sqr.size
res146: Int = 6
```

Detalhes de definição de classes

- Definição de construtores e atributos.
  - `class Square (val size : Int) {`
- Definição de métodos
  - `def area : Int = size * size`
  - ```
    def display : Unit = {
      println("Size = " + size)
      println("Area = " + area)
    }
    ```
    ```

Objetos Singleton

- Pode-se criar um objeto singleton utilizando a palavra **object**

```
> object Counter {
     |   var count = 0
     |
     |   def currentCount(): Long = {
     |     count += 1
     |     count
     |   }
     | }
defined module Counter

> Counter.currentCount
res153: Long = 1

> Counter.currentCount
res154: Long = 2
```

Traits

- Traits segue o conceito de polimorfismo
  - Os métodos e atributos são herdados pelas classes que herdam a Trait

```
> trait Geometric {
     |   def height : Int
     |   def area    : Int
     |   def perimeter : Int
     | }
defined trait Geometric
```

Usando Trait

- Para usar a trait, utilize as palavras-chave **extends** ou **with**

```
> class Square (val size : Int) extends Geometric {
     |            def area : Int = size * size
     |            def height : Int = size
     |            def perimeter : Int = size*4
     |            def display : Unit = {
     |               println("Size = " + size)
     |               println("Area = " + area)
     |            }
     |       }
defined class Square

> val sqr = new Square(6)
sqr: Square = $iwC$$iwC$Square@4f90cf86

> val geom : Geometric = sqr
geom: Geometric = $iwC$$iwC$Square@4f90cf86

> geom.area
res155: Int = 36
```

Classes Case

- Classes “amênicas”, mais ou menos, com atributos imutáveis.
  - Basicamente para armazenar valores estáticos
  - Inicialização simples
  - Comparação simples

```
> case class Pessoa (nome: String, genero: String, idade: Long)
defined class Pessoa

> val p1 = Pessoa(”Jonas", "M", 45)
> val p2 = Pessoa(”Claudia", "F", 50)
> val p3 = Pessoa(”Jonas", "M", 45)

> if (p1==p2) ”Iguais" else ”Nao sao iguais"
res0: String = Nao sao igais

> if (p1==p3) ”Iguais" else ”Nao sao iguais"
res1: String = Iguais
```

Pacotes e Imports

- Pacotes são formas de organizar e separar os objetos e classes utilizando namespaces.
- Para declarar um pacote:
  - `package com.mycompany.time`
- Utilize a instrução **`import`** para trazer os pacotes para o contexto corrente.

```
// Importar tudo dentro do pacote.
import com.mycompany.time._
// Importar apenas o Objeto Timepiece
import com.mycompany.time.Timepiece
```

Exemplo de Pacotes e Imports

```
// File Timepiece.scala
package com.mycompany.time;
import scala.math._;
trait TimePiece { /* … */ }
```

```
// File AlarmClock.scala
package com.mycompany.time;
import scala.math._;
class AlarmClock extends TimePiece { /* … */ }
```

```
// File Store.scala
package com.mycompany.products;
import com.mycompany.time.TimePiece

class Store {
  def timepieces : Array[TimePiece] = new Array[TimePiece](100)
}
```

Um programa Standalone em Scala

- É um Objeto com um método principal (main) que executa um script.
- Você compila o Objeto com o commando scalac
  > **scalac SimpleApp.scala**
- Você executa o Objeto com o comando scala
  > **scala SimpleApp**

```
object  SimpleApp {
  def main(args: Array[String]) {  // args são os parâmetros da linha de
                                   // comando
    for (cur <- args) println(cur)
  }
}
```

Resumo

- Scala é uma linguagem **funcional** e **O.O**
- O **Interpretador Scala** é uma boa opção para executar scripts simples
  - **REPL** (Read-Evaluate-Print Loop)
- Todos os dados dentro do Scala possuem **tipos**
  - Não necessáriamente precisam ser explicitos.
- Scala possue **Garbage Collection**
  - Objetos não utilizados são automáticamente reciclados da memória

Resumo

- Variáveis definidas com a instrução **`val`** são imutáveis
- Uma tupla é um grupo de objetos ordenados.
- **Tipos de coleções** no Scala
  - `Set, List, Map`
- O tipo **`Unit`** é usado para funções que não tem retorno

Conteúdos sobre Scala

- Book: **Programming in Scala**
  - By Martin Odersky (Scala language Designer), et. al
  - 1st edition online at: http://www.artima.com/pins1ed/

- Scala website: www.scala-lang.org
  - **Scala CheatSheet**: http://docs.scala-lang.org/cheatsheets/