



# Aceleração Global Dev

## Scala: o poder de uma linguagem multiparadigma

---

Ivan Pereira Falcão  
Expert Data/Cloud Engineer



# Objetivos da Aula

**1.** Conhecendo e compilando  
em Scala

**2.** A sintaxe do Scala

**3.** Orientação a objetos

# Requisitos Básicos

- ✓ Conhecimentos básicos de Shellscript
- ✓ Conhecimentos básicos de Linux
- ✓ Conhecimentos básicos de lógica de programação

# Parte 1: Conhecendo e compilando em Scala

Scala: o poder de uma linguagem multiparadigma

# Conhecendo o Scala

Scala combina programação orientada a objetos e programação funcional em uma linguagem concisa de alto nível.

- Scala roda encima da JVM (Java Virtual Machine), sendo assim, completamente interoperável com Java



# Instalando o Scala

Para instalar o open JDK:

- Para instalar o maven em distribuições Linux baseadas em apt-get:
  - `sudo apt-get install openjdk-8-jdk`
- Para instalar o maven em distribuições Linux baseadas em yum:
  - `sudo yum install java-1.8.0-openjdk`

# Instalando o Scala

Para instalar o Scala:

- Para instalar o maven em distribuições Linux baseadas em apt-get:
  - sudo apt-get install scala
- Para instalar o maven em distribuições Linux baseadas em yum:
  - wget <https://downloads.lightbend.com/scala/2.11.12/scala-2.11.12.rpm>
  - sudo rpm -i scala-2.11.12.rpm

# Apache Maven

- Apache Maven, ou Maven, é uma ferramenta de automação de compilação utilizada primariamente em projetos Java. Também é utilizada para construir e gerenciar projetos escritos em C#, Ruby, Scala e outras linguagens.
- O Maven utiliza um arquivo XML (POM) para descrever o projeto de software sendo construído, suas dependências sobre módulos e componentes externos, a ordem de compilação, diretórios e plug-ins necessários.

# Apache Maven

- Para instalar o maven em distribuições Linux baseadas em apt-get:
  - sudo apt-get install maven
- Para instalar o maven em distribuições Linux baseadas em yum:
  - sudo yum install maven

# Archetypes

Vamos utilizar o Maven para criação do nosso projeto. Utilizaremos o seguinte archetype para gerar as estruturas necessárias:

- Group: net.alchim31.maven
- ArtifactID: scala-archetype-simple
- Version: 1.6

Utilizar o comando:

```
mvn archetype:generate -DarchetypeGroupId=net.alchim31.maven  
-DarchetypeArtifactId=scala-archetype-simple  
-DarchetypeVersion=1.6
```

Adicionar as informações:

- GroupID;
- ArtfactID;
- Version;
- Package;

Se tudo estiver Ok, apenas confirme com “Y”

- Deletar o diretório src/test/scala/samples
- No arquivo pom.xml:
  - Remover a linha <arg>-make:transitive</arg>
  - Alterar a linha <scala.version>2.11.5</scala.version> para <scala.version>2.11.12</scala.version>

# Executando meu código

- Para gerar o projeto, entre no diretório contendo o arquivo pom.xml do projeto e:
  - Utilize o comando mvn package
- Para executar o programa:
  - scala -classpath target/{nome do jar}.jar {meu pacote}.{meu objeto}

Podemos criar um uberjar/fatjar:

- No pom, como último elemento dentro da tag plugins, adicionar:

```
<!-- Maven Shade Plugin -->
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <configuration>
    <createDependencyReducedPom>false</createDependencyReducedPom>
    <filters>
      <filter>
        <artifact>*:*</artifact>
      <excludes>
        <exclude>META-INF/*.SF</exclude>
        <exclude>META-INF/*.DSA</exclude>
        <exclude>META-INF/*.RSA</exclude>
      </excludes>
    </filter>
  </filters>
</configuration>
<executions>
  <execution>
    <phase>package</phase>
    <goals>
      <goal>shade</goal>
    </goals>
  <configuration>
```



- Agora podemos rodar nosso programa usando
  - `java -classpath target/{nome do fat jar}.jar {meu pacote}.{meu objeto}`
  - Com o plugin que instalamos o fat jar ficará: `{nome do jar}-jar-with-dependencies.jar`

# Parte 2: A sintaxe do Scala

Scala: o poder de uma  
linguagem multiparadigma

# Object

O tipo object armazena métodos que serão utilizados sem a necessidade de instancia-los. Para quem vem de java, object é o equivalente a um singleton. O objeto principal a ser chamado pelo scala **precisa ter o método “def main(args : Array[String])”**. Exemplo:

```
package com.everis

object App {

    def main(args : Array[String]) {
        println( "Hello World!" )
    }

}
```

## Comentários

Comentários em scala podem ser feitos de duas maneiras:

- Comentários linha a linha utilizando “//”:  
`//isso é um comentário!`
- Comentários de blocos utilizando “/\*” para iniciar o bloco e “\*/” para finalizar o bloco:  
`/* isso é  
 um bloco  
 de comentários  
 */`

## Println

O comando `println` é utilizado para imprimir na tela valores do tipo string.

```
println("Hello, James ")
```

```
println(s"1 + 1 = ${1 + 1}")
```

```
val name = "James"  
println(s"Hello, $name")
```

## var e val

Há duas maneiras de definir variáveis em scala:

O indicador **var** indica de fato variáveis que podem ser alteradas ao longo da execução do programa:

```
var minhaVariavel = "Valor inicial"  
println(minhaVariavel)  
minhaVariavel = "Valor alterado"  
println(minhaVariavel)
```

## var e val

Os indicadores **val** são definidos apenas no momento em que são criados, não podendo haver redefinição

```
val minhaVariavelVal = "Valor inicial"  
println(minhaVariavelVal)  
//O comando abaixo não irá funcionar  
minhaVariavelVal = "Valor inicial"
```

## Sobre tipos de variáveis

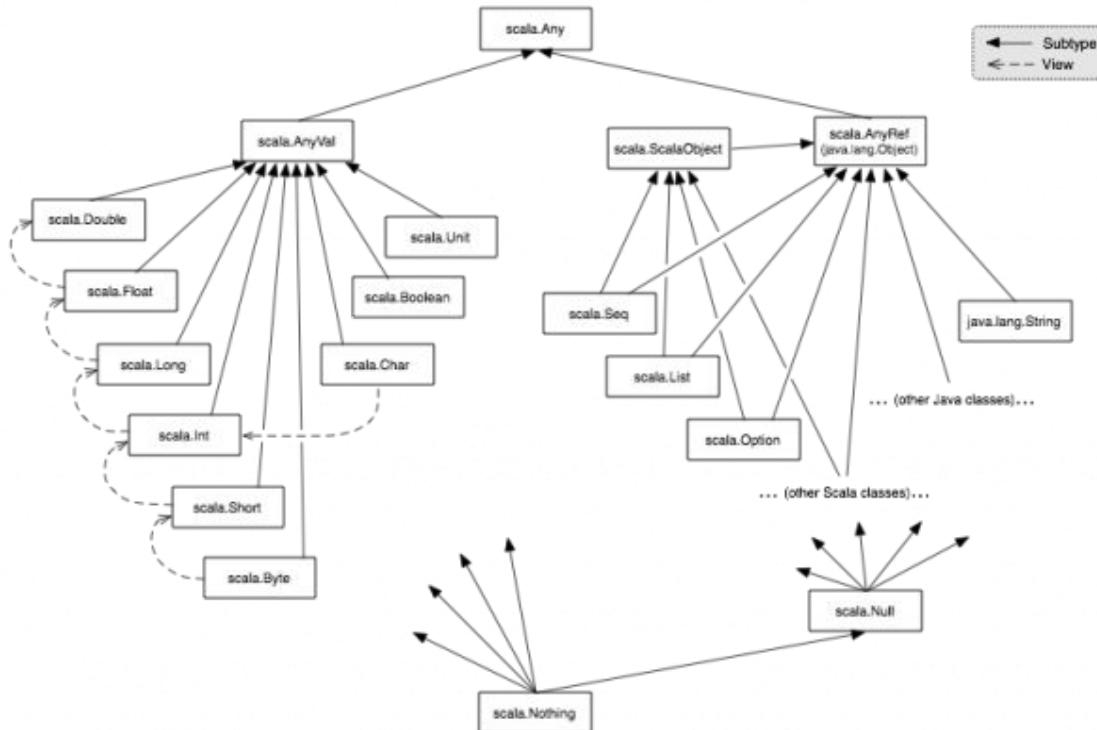
Scala, diferentemente de Python e Javascript, é uma linguagem tipada. Em geral, o próprio compilador do Scala se encarrega de inferir o tipo de uma variável, porém, para usufruirmos ao máximo da ferramenta, o ideal é indicarmos o tipo da variável que estamos configurando:

```
//O compilador infere que a variável é tipo String  
val minhaVariavelNaoTipada = "Valor inicial"
```

```
//O compilador espera que o valor seja do tipo String  
val minhaVariavelTipada: String = "Valor inicial"
```



# Sobre tipos de variáveis



# Sobre tipos de variáveis

Os objetos do Scala contém métodos próprios para conversão entre os tipos básicos:

```
val varialvellinteira: Int = 30
```

```
val variavelDouble: Double = 3.14
```

```
val variavelString: String = "35"
```

```
val novaString: String = varialvellinteira.toString
```

```
val novoFloat: Double = varialvellinteira.toDouble
```

```
val novolnteiro1: Int = variavelDouble.toInt
```

```
val novaString1: String = variavelDouble.toString
```

```
val novolnteiro2: Int = variavelString.toInt
```

```
val novaFloat2: Double = variavelString.toDouble
```

# Operadores

As operações abaixo podem ser executadas apenas com tipos numéricos (Byte, Short, Int, Long, Float, Double):

Operator	Description
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão inteira

# Operadores

As operações abaixo podem ser executadas apenas com tipos numéricos (Byte, Short, Int, Long, Float, Double):

```
val variavel1 = 8  
val variavel2 = 3
```

```
val soma: Int = variavel1 + variavel2  
val subtr: Int = variavel1 - variavel2  
val mult: Int = variavel1 * variavel2  
val divi: Float = variavel1 / variavel2  
val resto :Int = variavel1 % variavel2
```

# Operadores

As operações abaixo se referem a operações lógicas

Operator	Description
<code>==</code>	Operador de Igual. Retorna True se verdadeiro e False em caso negativo
<code>!=</code>	Operador de Diferente. Retorna True se verdadeiro e False em caso negativo
<code>&gt;</code>	Operador de Maior. Retorna True se verdadeiro e False em caso negativo
<code>&lt;</code>	Operador de Menor. Retorna True se verdadeiro e False em caso negativo
<code>&gt;=</code>	Operador de Maior ou igual. Retorna True se verdadeiro e False em caso negativo
<code>&lt;=</code>	Operador de Menor ou igual. Retorna True se verdadeiro e False em caso negativo
<code>!</code>	Operador lógico NOT. Inverte a lógica de uma operação boolena
<code>  </code>	Operador lógico OR. Retorna True se um dos dois operandos forem verdadeiros
<code>&amp;&amp;</code>	Operador lógico AND. Retorna True se os dois operandos forem verdadeiros

# Condicionais

O Scala trabalha com condicionais de maneira muito semelhante a outras linguagens:

```
//Exemplo de if básico
```

```
var condicional = 1
if (condicional == 1) {
    println("Variavel igual a um")
} else {
    println("Variavel não é igual a um")
}
```

```
//Exemplo de if concatenado
```

```
var condicional = 4
if (condicional == 1) {
    println("Variavel igual a um")
} else if(condicional > 3) {
    println("Variavel maior que três")
}
```

# Condicionais

O Scala trabalha com condicionais de maneira muito semelhante a outras linguagens:

```
//Exemplo de match
val matchVal = 25
matchVal match {
    case 1 => println("Domingo")
    case 2 => println("Segunda")
    case 3 => println("Terça")
    case 4 => println("Quarta")
    case 5 => println("Quinta")
    case 6 => println("Sexta")
    case 7 => println("Sabado")

    case _ => println("Caso inesperado: " + whoa.toString)
}
```

# Iteráveis

Um dos tipos mais comuns de objetos Iteráveis são os Arrays:

```
//Array Vazio  
val meuArray:Array[String] = Array[String]()
```

```
//Tamanho (length) do Array  
println(s"Tamanho do meu array: ${meuArray.length}")
```

```
//Array com três elementos  
val meuArray2:Array[String] = Array[String]("Valor 1", "Valor 2", "Valor 3")
```

```
//Atualizando seu Array  
meuArray2.update(0,"Novo Valor2")
```

```
//Visualizando dados do Array  
println(s"Visualizando dados do meu Array: ${meuArray2(0)}")
```

# Iteráveis

Arrays são objetos estáticos em Scala; caso necessite de uma estrutura mais maleável, o ideal é utilizar o ArrayBuffer:

```
//Importando o componente Array Buffer
import scala.collection.mutable.ArrayBuffer
```

```
//Instanciando e adicionando elementos
var frutas = ArrayBuffer[String]()
frutas += "Maçã"
frutas += "Banana"
frutas += "Laranja"

println(s"Eu gosto de ${frutas(1)} ")
```



## Iteráveis

Listas são um outro exemplo de iteráveis estáticos. Diferentemente do Array, listas não contém métodos de update:

```
//Instanciando uma lista
val minhaLista = List[Int](1, 2, 3, 4)
println(s"Minha Lista contém ${minhaLista(1)} na posição 1")
```

# Iteráveis

Para o equivalente dinâmico da lista temos o ListBuffer:

```
//Importando o componente ArrayBuffer  
import scala.collection.mutable.ListBuffer
```

```
//Instanciando e adicionando elementos  
val listaMutavel = ListBuffer[String]()  
listaMutavel += "Maçã"  
listaMutavel += "Banana"  
listaMutavel += "Laranja"
```

```
println(s"Eu gosto de ${listaMutavel(2)} ")
```

# Iterando objetos

No Scala temos várias maneiras de Iterar objetos. Um dos mais tradicionais é o laço for:

```
//Objeto Iteravel
val iteravel:Array[String] = Array[String]("Valor 1", "Valor 2", "Valor 3", "Valor 4", "Valor 5")
```

```
//Iterando os valores diretamente
for (valores <- iteravel) {
    println(s"Valor retornado: $valores")
}
```

```
//Iterando valores com um índice
for (i <- 0 to iteravel.length - 1) {
    println(s"Os valores do objeto são: ${iteravel(i)}")
}
```

# Iterando objetos

No Scala temos várias maneiras de Iterar objetos. Outro método tradicional é o while:

```
//Objeto Iteravel
val iteravel:Array[String] = Array[String]("Valor 1", "Valor 2", "Valor 3", "Valor 4",
"Valor 5")
```

```
//Iterando via while
var i = 0
while (i < iteravel.length) {
  println(s"Os valores do objeto usando while: ${iteravel(i)}")
  i += 1
}
```

# Iterando objetos

Quando estamos falando de iterar objetos a melhor maneira de iterarmos um objeto é através dos métodos próprios foreach e map:

```
//Objeto Iteravel
val iteravel:Array[String] = Array[String]("Valor 1", "Valor 2", "Valor 3", "Valor 4", "Valor 5")
```

```
//Iterando via foreach
iteravel.foreach(value => {
  println(s"Imprimindo valores com foreach $value")
})
```

```
//Iterando via map
iteravel.map(value => {
  println(s"Imprimindo valores com foreach $value")
})
```

# Iterando objetos

A principal diferença entre foreach e map é que map retorna um novo objeto iterável com as transformações indicadas dentro da função implícita, enquanto foreach apenas executa a função, sem retornar valores

```
//Trabalhando com foreach e map em uma lista de tuplas
val listaDeTuplas = List[(String, Int)](("chave 1", 25), ("chave 2", 30), ("chave 3", 35))
```

```
//Imprimindo valores com foreach
listaDeTuplas.foreach(tupla => {
    println(s"Chave: ${tupla._1} Valor ${tupla._2}")
})
```

# Iterando objetos

A principal diferença entre foreach e map é que map retorna um novo objeto iterável com as transformações indicadas dentro da função implícita, enquanto foreach apenas executa a função, sem retornar valores

```
//Criando uma nova Lista utilizando map
val novaListaDeTuplas: List[(Int, String)] = listaDeTuplas.map(tupla => {
    (tupla._2 * 3, tupla._1)
})
```

```
//Usando a função print de maneira implícita
novaListaDeTuplas.foreach(println)
```

## Filtragem

Uma outra propriedade útil dos objetos iteráveis do Scala são as funções filter e filternot:

```
//Filtrando nossas listas
val listaFiltrada = novaListaDeTuplas.filter(tupla => tupla._1 > 75)
listaFiltrada.foreach(println)
```

```
//Filtragem invertida
val listaFiltradaInvertida = novaListaDeTuplas.filterNot(tupla =>
tupla._1 > 75)
listaFiltradaInvertida.foreach(println)
```

# Reduções

Baseado no modelo de map reduce, também podemos utilizar a função reduce do Scala:

```
//Exemplo de redução
val listaReduceLeft = List[Int](5, 10, 15, 20)
val reduced = listaReduceLeft.reduce((valor1, valor2) => valor1 + valor2)

println(s"Valor Reduzido: $reduced")
```

```
//Utilizando reduce para encontrar o menor valor
val minhaListaReduceSide: List[Int] = List(5, 6, 7, 8 ,9, 3, 25)
val resultadoReduceMin = minhaListaReduceSide.reduce((variavel1, variavel2) =>
{variavel1 min variavel2})
println(s"Resultado Reduce Right Max: $resultadoReduceMin")
```

## Group By

O groupby é uma função que agrupa os elementos de um iterável, baseando-se em um dos elementos internos:

```
//Exemplo de groupby
val listaParaAgrupar: List[(String, Int)] = List(("Valor 1", 50), ("Valor 2",
40), ("Valor 1", 15), ("Valor 2",55), ("Valor 3", 10 ))

val resListAgrup = listaParaAgrupar
    .groupBy(listTupl => listTupl._1)

println(s"Resultado Lista Agrupada: $resListAgrup")
```

## Sort By

Sort By é uma função que ordena os elementos da lista

```
//Exemplo de sortBy
```

```
val listaParaOrdenar : List[(String, Int)] = List(("Valor 25", 50), ("Valor 32",  
40), ("Valor 5", 15), ("Valor 1", 55), ("Valor 2", 10 ))
```

```
val resListOrdenada = listaParaOrdenar  
.sortBy(listTupl => listTupl._1)
```

```
println(s"Resultado Lista Ordenada: $resListOrdenada")
```

# Funções

Funções em Scala podem ser definidas da seguinte forma:

```
//Exemplo de função que não retorna valores
def minhaFunçãoDelImpressão(inteiro: Int): Unit = {
    println(s"O dobro de $inteiro é ${inteiro * 2}")
}
```

```
minhaFunçãoDelImpressão(20)
```

```
//Exemplo de função que retorna valores
def minhaFunçãoQueRetornaODobro(inteiro: Int): Int = {
    inteiro * 2
}
```

```
println(s"O valor retornado é ${minhaFunçãoQueRetornaODobro(20)}")
```

# Funções

Também podemos declarar funções dentro de variáveis:

```
//Exemplo de funções em variáveis
val minhaFuncaoEmVariavel: Int => Unit = (valor: Int) => {println(s"Valor
da minha função em variável: ${valor * 4}")}

minhaFuncaoEmVariavel(34)
```

```
//Exemplo de funções em variáveis retornando valores
val minhaFuncaoRetornandoMultPi: Int => Double = (valor: Int) => {valor
* 3.14}

println(s"Resultado da minha função implicita:
${minhaFuncaoRetornandoMultPi(34)}")
```

# Parte 3: Orientação a objetos

Scala: o poder de uma  
linguagem multiparadigma

# Orientação a objetos em Scala

Em Scala, uma classe é definida da seguinte maneira:

```
package com.everis
```

```
class MinhaNovaClasse {
```

```
}
```

```
//Instanciando minha classe
```

```
val minhaClasseInstaciada: MinhaNovaClasse = new MinhaNovaClasse
```

# Orientação a objetos em Scala

Construtores são definidos no corpo da classe:

```
package com.everis

class MinhaNovaClasse(nome: String) {
    println(s"Nome: $nome")
}

//Instanciando minha classe
val minhaClasseInstaciada: MinhaNovaClasse = new
MinhaNovaClasse("Ivan")
```

# Orientação a objetos em Scala

Em Scala uma classe pode ter multiplos construtores:

```
package com.everis
```

```
class MinhaNovaClasse(var nome: String, var idade: Int, var altura: Int) {  
    def this(nome: String) {  
        this(nome, 0, 0)  
    }  
  
    def this(nome: String, idade: Int) {  
        this(nome, idade, 0)  
    }  
    println(s"Nome: $nome Idade: $idade Altura: $altura")  
}
```

# Orientação a objetos em Scala

Em Scala uma classe pode ter multiplos construtores:

```
//Instanciando minha classe
val minhaClasseInstaciada1: MinhaNovaClasse = new
MinhaNovaClasse("Ivan")
val minhaClasseInstaciada2: MinhaNovaClasse = new
MinhaNovaClasse("Ivan", 29)
val minhaClasseInstaciada3: MinhaNovaClasse = new
MinhaNovaClasse("Ivan", 29, 175)
```

# Orientação a objetos em Scala

Métodos de classe são declarados da mesma forma que funções:

```
package com.everis
```

```
class MinhaNovaClasse(nome: String, idade: Int, altura: Int) {
```

```
    def meuMétodo: Unit = {
        println(s"Meu nome é: $nome")
    }
```

```
}
```

```
//Instanciando minha classe e usando o método
```

```
val minhaClasselInstaciada: MinhaNovaClasse = new MinhaNovaClasse("Ivan",
29, 175)
minhaClasselInstaciada.meuMétodo
```

# Orientação a objetos em Scala

Variáveis e métodos podem ser públicos, privados e protegidos. Garantindo ou não a visualização dos mesmos:

```
package com.everis
class MinhaNovaClasse(nome: String, idade: Int, altura: Int) {

    protected val idadeProtegida = idade

    def meuMetodoPublico: Unit = {
        qualMeuNome()
    }

    private def qualMeuNome(): Unit = {
        println(s"Meu nome é: $nome")
    }
}
```

# Orientação a objetos em Scala

Variáveis e métodos podem ser públicos, privados e protegidos. Garantindo ou não a visualização dos mesmos:

```
//Instanciando minha classe e usando o método público
val minhaClasselInstaciada: MinhaNovaClasse = new MinhaNovaClasse("Ivan",
29, 175)
minhaClasselInstaciada.meuMetodoPublico
```

# Orientação a objetos em Scala

Herança em Scala pode ser representada da seguinte maneira:

```
package com.everis
```

```
class ClassePai {  
    def metodoPaiPublico: Unit = {  
        println("Método pai publico")  
    }
```

```
protected def metodoPaiProtegido: Unit = {  
    println("Método pai protegido")  
}
```

```
private def metodoPaiPrivado: Unit = {  
    println("Método pai protegido")  
}
```

# Orientação a objetos em Scala

Herança em Scala pode ser representada da seguinte maneira:

```
package com.everis
```

```
class ClasseFilha extends ClassePai {  
    def metodoFilhoPublico: Unit = {  
        super.metodoPaiProtegido  
        super.metodoPaiPublico  
    }  
}
```

```
val minhaHeranca = new ClasseFilha  
minhaHeranca.metodoPaiPublico  
minhaHeranca.metodoFilhoPublico
```

# Orientação a objetos em Scala

Links úteis:

- [https://docs.scala-lang.org/?\\_ga=2.219152154.265858086.1612118441-215605817.1612018441](https://docs.scala-lang.org/?_ga=2.219152154.265858086.1612118441-215605817.1612018441)
- <https://alvinalexander.com/scala>
- [https://www.amazon.com.br/Learning-Scala-Practical-Functional-Programming-ebook/dp/B00QW1RQ94/ref=sr\\_1\\_3?\\_\\_mk\\_pt\\_BR=%C3%85%C3%85%C5%BD%C3%95%C3%91&dchild=1&keywords=Scala+language&qid=1612117681&sr=8-3](https://www.amazon.com.br/Learning-Scala-Practical-Functional-Programming-ebook/dp/B00QW1RQ94/ref=sr_1_3?__mk_pt_BR=%C3%85%C3%85%C5%BD%C3%95%C3%91&dchild=1&keywords=Scala+language&qid=1612117681&sr=8-3)

# Dúvidas?

Scala: o poder de uma  
linguagem multiparadigma