

Introdução à Linguagem Scala

Paradigmas de Linguagens de Programação

Gabriel Viana de Almeida

Ausberto S. Castro Vera

22 de junho de 2023



Disciplina: Paradigmas de Linguagens de Programação 2023

Linguagem: Scala

Aluno: *Gabriel Viana de Almeida*

Ficha de avaliação:

Aspectos de avaliação (requisitos mínimos)	Pontos
Introdução (Máximo: 01 pontos) <ul style="list-style-type: none"> • Aspectos históricos • Áreas de Aplicação da linguagem 	
Elementos básicos da linguagem (Máximo: 01 pontos) <ul style="list-style-type: none"> • Sintaxe (variáveis, constantes, comandos, operações, etc.) • Cada elemento com exemplos (código e execução) 	
Aspectos Avançados da linguagem (Máximo: 2,0 pontos) <ul style="list-style-type: none"> • Sintaxe (variáveis, constantes, comandos, operações, etc.) • Cada elemento com exemplos (código e execução) • Exemplos com fonte diferenciada (listing) 	
Mínimo 5 Aplicações completas - Aplicações (Máximo : 2,0 pontos) <ul style="list-style-type: none"> • Uso de rotinas-funções-procedimentos, E/S formatadas • Uma Calculadora • Gráficos • Algoritmo QuickSort • Outra aplicação • Outras aplicações ... 	
Ferramentas (compiladores, interpretadores, etc.) (Máximo : 1,0 pontos) <ul style="list-style-type: none"> • Ferramentas utilizadas nos exemplos: pelo menos DUAS • Descrição de Ferramentas existentes: máximo 5 • Mostrar as telas dos exemplos junto ao compilador-interpretador • Mostrar as telas dos resultados com o uso das ferramentas • Descrição das ferramentas (autor, versão, homepage, tipo, etc.) 	
Organização do trabalho (Máximo: 01 ponto) <ul style="list-style-type: none"> • Conteúdo, Historia, Seções, gráficos, exemplos, conclusões, bibliografia • Cada elemento com exemplos (código e execução, ferramenta, nome do aluno) 	
Uso de Bibliografia (Máximo: 01 ponto) <ul style="list-style-type: none"> • Livros: pelo menos 3 • Artigos científicos: pelo menos 3 (IEEE Xplore, ACM Library) • Todas as Referências dentro do texto, tipo [ABC 04] • Evite Referências da Internet 	
Conceito do Professor (Opcional: 01 ponto)	
<p style="text-align: right;">Nota Final do trabalho:</p>	

Observação: Requisitos mínimos significa a metade dos pontos

Copyright © 2023 Gabriel Viana de Almeida e Ausberto S. Castro Vera

UENF - UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE DARCY RIBEIRO

CCT - CENTRO DE CIÊNCIA E TECNOLOGIA

LCMAT - LABORATÓRIO DE MATEMÁTICAS

CC - CURSO DE CIÊNCIA DA COMPUTAÇÃO

Primeira edição, Abril 2023

Sumário

1	Introdução a Linguagem Scala	7
1.1	História	7
1.2	Características	8
1.3	Java Virtual Machine	8
1.4	Simple Build Tool	9
1.5	Aplicações	10
1.5.1	Big Data	10
1.5.2	Finanças	11
1.5.3	Aprendizado de Máquinas	11
2	Conceitos básicos da Linguagem Scala	13
2.1	Variáveis	13
2.1.1	Val	13
2.1.2	var	14
2.2	Tipos de dados	14
2.2.1	String	14
2.2.2	Char	14
2.2.3	Int	15
2.2.4	Long	15
2.2.5	Short	15
2.2.6	Byte	15
2.2.7	Float	15
2.2.8	Double	16
2.2.9	Boolean	16

2.3	Operadores e expressões em Scala	16
2.3.1	Operadores aritméticos	16
2.3.2	Operadores de comparação	17
2.3.3	Operadores lógicos	18
2.4	Entrada e Saída	19
2.4.1	Saída	19
2.4.2	Entrada	19
2.5	Condicionais	20
2.5.1	IF	20
2.5.2	match	21
2.6	Estruturas de repetição	21
2.6.1	while	21
2.6.2	For	21
2.6.3	Foreach	22
3	Programação em Scala	23
3.1	Funções	23
3.1.1	Função anônima	24
3.2	Classe	24
3.3	Listas	25
3.3.1	Operações com listas	25
3.4	Array	26
3.5	Vector	26
3.6	Tuplas	27
3.7	Set	28
3.8	Maps	29
4	Aplicações da Linguagem Scala	33
4.1	Calculadora	33
4.2	Cilindro	36
4.3	Parametrização de Retas	39
4.4	SelectionSort	41
4.5	QuickSort	44
4.6	Soma de Matrizes	47
4.7	Aplicações Biologia	51
4.8	Aplicação física	55
5	Ferramentas existentes e utilizadas	57
5.1	Visual Studio Code	57
5.2	Scastie	59
6	Considerações Finais	63
	Bibliografia	67

1. Introdução a Linguagem Scala

Esse capítulo iremos apresentar a história e aplicações da linguagem Scala.

1.1 História

A Scala é a contração das palavras Scalable Language. Seu criador é **Martin Odersky** atualmente professor School of Computer and Communication Sciences na École Polytechnique Fédérale de Lausanne (EPFL), localizada na Suíça. Quando Martin era professor universitário em Karlsruhe, Alemanha, em 1988, ele ouviu falar da criação do Java ainda em fase alfa. Ele era interessado pela programação funcional e se juntou com Phil Wadler, um dos criadores de Haskell, e lançaram a linguagem Pizza utilizando o ambiente do Java e algumas ideias da programação funcional.

“Phil Wadler e eu decidimos pegar algumas ideias da programação funcional e levá-las para o espaço Java. Esse esforço se tornou uma linguagem chamada Pizza, que tinha três recursos da programação funcional: genéricos, funções de ordem superior e correspondência de padrões. A distribuição inicial do Pizza foi em 1996, um ano depois do lançamento do Java. Foi moderadamente bem-sucedido, mostrando que era possível implementar recursos de linguagem funcional na plataforma JVM.”[[VS09](#)]

Tempo depois, entraram em contato com a **Sun Microsystems**, a criadora do Java, e começaram a trabalhar juntos. Em 1997, desenvolveram GJ (Generic Java) que se tornou parte do Java 5 seis anos depois.

Foi na experiência do desenvolvimento de ambos projetos que ele se sentiu desapontado com algumas restrições do Java.

"Durante a experiência com Pizza e GJ, às vezes me senti frustrado, porque o Java é uma linguagem existente com restrições muito rígidas. Como resultado, não pude fazer muitas coisas do jeito que gostaria de fazer - do jeito que eu estava convencido de que seria o certo. Então, depois daquele tempo, quando essencialmente o foco do meu trabalho era tornar o Java melhor, decidi que era hora de dar um passo atrás.

Eu queria começar do zero e ver se poderia projetar algo melhor que o Java. Mas ao mesmo tempo, sabia que não poderia começar do zero. Eu tinha que me conectar a uma infraestrutura existente, porque, caso contrário, seria impraticável sair do zero sem bibliotecas, ferramentas e coisas assim."[VS09]

Portanto, surgiu a ideia de criar algo diferente do Java, mas que utilizasse ainda suas bibliotecas e a Java Virtual Machine (JVM), e fosse capaz de poder lidar com alto volume de requerimentos. Formou um pequeno grupo de pesquisadores, na atual faculdade onde trabalha, a EPFL. Em 2002 iniciaram de fato o desenvolvimento da Scala, e foi lançada ao público em 2003; Em 2006 saiu sua versão 2.0 e, mais recentemente, foi lançada a versão 3.0 em março de 2021.

1.2 Características

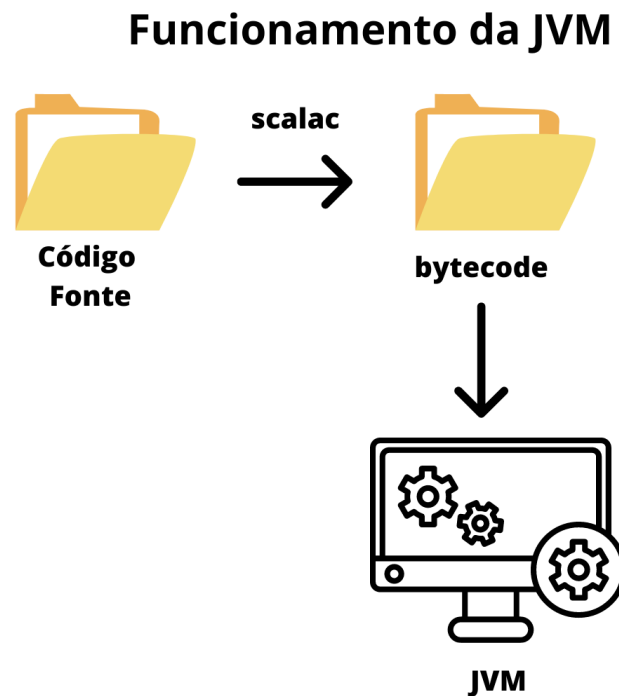
Suas principais características são:

- **Tipagem estática:** [OMS⁺21] diz que umas das vantagens da Scala é justamente o seu tipo precisar ser explicitamente declarado. No ato da compilação, é verificado o código fonte, se houver erros não executará. Isso torna a linguagem menos propensa a erros.
- **Multiparadigmas:** [Sfr21] argumenta que, Scala é uma linguagem híbrida: Orientada a objetos e funcional.
[Sfr21] ainda afirma as vantagens e desvantagens de ambos paradigmas. A orientação a objetos tende ser rápido, porém com maior probabilidade a erros, pois usa estados mutáveis. Com isso, seu programa realocará sua memória sempre que ocorrer uma mudança, isso poderá causar possíveis erros de dados.
O paradigma funcional, por sua vez, não é tão performática quando o paradigma orientado a objetos, porém é mais seguro, já que seus dados não mudarão. Seu código é mais reutilizável e fácil de ler. Porém, sua desvantagem é a maior utilização de memória porque não permite atualizar as variáveis, apenas recria-las. É uma das melhores ferramentas para big data e problemas de concorrência
- **Multiplataforma** Scala é multiplataforma graças ao JVM. [Sfr21] explica que é uma máquina virtual que permite rodar código em qualquer sistema operacional.
- **Sintaxe:** Possui uma sintaxe simples comparada com outras linguagens como C, java, C++. [OMea06] afirma que sua sintaxe usa grande parte de Java e C#.
- **Compatibilidade com Java:** [Sfr21] argumenta que Scala permite usar bibliotecas já conhecidas do Java.

1.3 Java Virtual Machine

O que é a JVM? [Sfr21] afirma que é uma máquina virtual cujo o código fonte é compilado pela *scalac* para *bytecode* que será executado pela JVM, permitindo rodar o código em qualquer arquitetura de hardware, uma grande vantagem da linguagem. Ela possui uma filosofia "Escreva uma vez, rode em qualquer lugar". A seguir uma ilustração do processo.

Figura 1.1: Funcionamento da JVM. Fonte: Autor

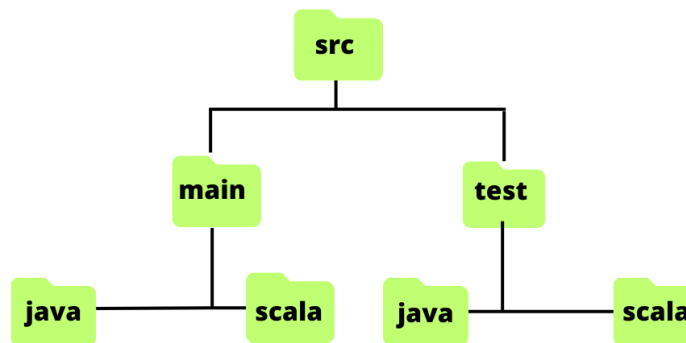


A JVM possui *garbage collector* para controlar a memória, verificando se não está mais sendo utilizada. Se for o caso, ela a remove afim de impulsionar a performance, diferente de linguagens como C, que precisam fazer esse controle manualmente. Ainda existe o conceito Just in Time (JIT), utilizada para compilar os códigos que são mais frequentes para linguagem de máquina. Por exemplo, a JIT conta quantas vezes uma função foi utilizada, se ultrapassar um certo número, ele a transformar para código de máquina sendo executada pelo processador diretamente.

1.4 Simple Build Tool

[Sfr21] ressalta que na linguagem Scala nós temos Simple Build Tool (SBT) que é a ferramenta mais popular para criar, gerenciar, executar e compilar projetos em Scala. Ela facilita na produção do código, deixando-o organizado em pastas. É possível ver uma versão simplificada na Figura 1.2

Figura 1.2: Hierarquia. Fonte: Autor



Nota-se que existe uma espécie de hierarquia. A pasta **src** é sua raiz, ela contém os códigos do projeto. A pasta **main** é dividida em duas subpastas, **scala** e **java**, que contém os códigos escritos nas linguagens Scala e Java, respectivamente. Do outro lado, temos a pasta **test**, considerada como uma pasta opcional, irá conter os testes da aplicação. Por fim, dentre dele também há duas pastas; a **scala** irá contar com seu código de teste em Scala, **java** irá conter seu código teste em Java.

1.5 Aplicações

Scala é usada nas seguintes aplicações, nesse livro destacaremos três, sendo elas:

1.5.1 Big Data

Por ser multiparadigma, Scala permite criar códigos mais seguros e legíveis. Como dito anteriormente, ela foi criada justamente para lidar com alto volume de dados.

A Scala tem um papel importante na área de big data, principalmente graças a ferramenta **Apache Spark** que possibilita processar esses dados.

A JVM permite a Scala ter um desempenho significativo, tornando-a mais rápida que inúmeras outras linguagens. Isso auxilia na sua performance em geral.

Figura 1.3: Big Data

Figura 1.4: Disponível em: [PNG WING](#)

1.5.2 Finanças

O mercado financeiro exige segurança e confiabilidade. A Scala prover de todas essas funções, graças a sua tipagem estática e permitir adotar o paradigma funcional, diminuindo assim os possíveis erros.

Além disso, há o fator dos frameworks da Scala tais como: Spark, dito anteriormente, e o **Akka**, utilizada em sistemas distribuídos e escaláveis, ainda é possível utilizar frameworks do Java, tornando-a muito completa.

Outro fator, é a escalabilidade da Scala. Ela foi projetada para lidar com alto volume de dados, ou seja, fundamental no mercado financeiro. É possível utilizá-la, por exemplo, para análise de dados financeiros.

Figura 1.5: Disponível em: [PNG WING](#)

1.5.3 Aprendizado de Máquinas

Sua tipagem a torna segura, impedindo que erros ocorram, o que é fundamental para o aprendizado de máquinas, uma área que lida com grandes quantidades de dados ao mesmo tempo.

Reaproveitar as bibliotecas do Java é outro fator de benefício para machine learning. O Java possui inúmeras bibliotecas já consagradas, e utilizá-las em Scala é um enorme benefício.

Scala permite a utilização do framework **Apache Spark MLlib**, que facilita o aprendizado de máquinas, sendo um dos principais frameworks para esse feito, ainda existe o **ScalaNLP** disponibilizando ferramentas para modelagem estática e linguagem neural. Juntamente com as características de tipagem da Scala e seu paradigma, a torna segura e confiável de testar, além de sua simples sintaxe.

Figura 1.6: Disponível em: [PNG WING](#)



2. Conceitos básicos da Linguagem Scala

Nesse capítulo estudaremos os conceitos básicos da linguagem. Veremos os tipos de variáveis, tipos de dados, como realizar operações entre eles, entrada e saída, condicionais e repetições.

2.1 Variáveis

Uma variável serve para armazenar valores que serão utilizados no programa. Para declarar uma variável, primeiro definimos o tipo da variável (`var` ou `val`), seu nome, o tipo de dado e, por fim, seu valor. Exemplo:

```
// Assim que definimos variavel
val nome: Tipo = valor
```

Um ponto importante é que o tipo é opcional, mas definir o tipo da variável pode tornar o código mais fácil de compreender e ajudar a evitar erros de tipagem. O "=" pode ser lido como "nome recebe valor". Iremos aprofundar agora em quais tipos de variáveis existem em Scala.

Em Scala nós temos dois tipos de variáveis: as imutáveis [`val`] e mutáveis [`var`].

2.1.1 Val

Com base no [Wam21] o `val` é utilizado para dados imutáveis. Uma vez declarada, não pode ser mudada. Veremos um exemplo.

```
scala> val msg = "Bom dia!"
var msg: String = Bom dia
```

No primeiro exemplo, estamos criando uma variável imutável. Nota-se que não é necessário definir **msg** como `string`, pois o compilador consegue inferir qual tipo é. Se tentarmos redefinir um novo valor a variável **msg** teremos um erro:

```
-- [E052] Type Error:
1 |msg = "Boa tarde!"
  |~~~~~
```

```
| Reassignment to val msg
|
| longer explanation available when compiling with
| '-explain'
1 error found
```

2.1.2 var

[Wam21] também fala sobre a variável `var` que permite seus dados sejam alterados em qualquer momento no programa. Exemplo:

```
scala> var msg = "Eu posso ser mudada"
var msg: String = Eu posso ser mudada

scala> msg = "Eu fui mudada"
msg: String = Eu fui mudada
```

No primeiro criamos um variável `msg` que recebe uma mensagem. Logo em seguida, alteramos seu valor.

2.2 Tipos de dados

Iremos falar agora sobre os dados básicos presente na linguagem Scala. Ele se refere a qual tipo de valor uma variável irá armazenar.

2.2.1 String

[OMS⁺21] diz que strings são uma sequência de caracteres. Elas podem conter frases, textos, números.

```
scala> var name: String = "Jorel"
var name: String = Jorel

val res0: String = Jorel
```

Nesse código, criamos uma variável `name` utilizando `var`, em seguida definimos seu tipo como `String` e atribuímos "Jorel" como seu valor. Note, a definição de `String` não é necessário, uma vez que a Scala consegue interpretar que o resultado entre aspas duplas é uma string, mas pode-se colocar para deixar o código mais claro de ler.

```
scala> name = "Tiago"
name: String = Tiago
```

No exemplo acima trocamos a variável, antigamente chamada de "Jorel", e a definimos agora como Tiago.

Você teria problemas para exibir sua string caso ela possuía aspas duplas dentro dela. Para isso, é possível iniciar uma string com aspas triplas (""), com isso não restringiria a mensagem.

```
scala> println("""A placa diz "Limite 60 km/h" """)
A placa diz "Limite 60 km/h"
```

2.2.2 Char

[OMS⁺21] afirma que em Scala, assim como várias outras linguagens, o `char` representa apenas caractere UNICODE. É representado por aspas simples (').

```
scala> val letra: Char = 'H'  
val letra: Char = H
```

Definimos acima um char que seu valor é representado por 'H'. Lembre-se, precisa estar em aspas simples. Outro fator é que a Scala entende os valores como UNICODE, portanto 'H' e 'h' não são a mesma coisa. O código de cada um em UNICODE são, respectivamente, U+0048 e U+0068.

Você pode fornecer um valor de Char utilizando seu código em UNICODE.

```
scala> val F: Char = '\u0046'  
val F: Char = F
```

2.2.3 Int

[OMS⁺21] ressalta que int se referem ao tipos de dados de números inteiros positivos ou negativos. É permitido realizar operações aritméticas.

```
scala> var num: Int = 10  
var num: Int = 10
```

No código acima, definimos um inteiro com o valor de 10. Podemos realizar operações com eles, tais como: contadores de loops, índices de listas, e etc... Extremamente importante e constantemente utilizados. Em Scala, números inteiros suportam valores até 32 bits (-2^{31} à $2^{31} - 1$).

2.2.4 Long

O princípio é o mesmo do tipo Int, entretanto, ele suporta valores até 64 bits. Segundo [OMS⁺21] para declarar um Long, é colocar um sufixo L ao final do valor para dizer ao compilador o seu tipo. Valores long: (-2^{63} à $2^{63} - 1$).

```
scala> val myNumber = 130654789521L  
val myNumber: Long = 130654789521
```

2.2.5 Short

O short, como o nome diz, é para valores menores do que inteiro. [Wam21] menciona que pode-se armazenar até (-2^{15} à $2^{15} - 1$).

```
scala> var myShort: Short = 52  
var myShort: Short = 52
```

2.2.6 Byte

Para armazenar valores ainda menores que o short, [Wam21] afirma que pode ser utilizado o Byte que consiste em valores até (-2^7 à $2^7 - 1$).

```
scala> val myByte: Byte = 1  
val myByte: Byte = 1
```

2.2.7 Float

O float é para valores reais de ponto flutuante de até 32 bits. Muito usados para cálculos de resultados não exatos. [OMS⁺21] argumenta que, assim como no 'Long', é necessário deixar o sufixo 'F' no final para que o compilador não interprete como Double.

```
scala> val media: Float = 6.45f  
val media: Float = 6.45
```

2.2.8 Double

[OMS⁺21] discute que o double é para valores reais de ponto flutuante de até 64 bits, permitindo maior precisão.

```
scala> val PI: Double = 3.141592653589793
val PI: Double = 3.141592653589793
```

2.2.9 Boolean

[Wam21] ressalta que boolean são tipos lógicos aceitam apenas dois valores: true e false. São de extrema importância para operações lógicas.

```
scala> val verdade: Boolean = true
val verdade: Boolean = true

scala> val mentira: Boolean = false
val mentira: Boolean = false
```

2.3 Operadores e expressões em Scala

Veremos nessa secção como realizar operações aritmética, comparações e lógicas.

2.3.1 Operadores aritméticos

São responsáveis por realizar operações tais como: Adição, subtração, divisão e etc...

- + : Adição.
- - : Subtração.
- * : Multiplicação.
- / : Divisão.
- % : Resto da divisão.

Esses são os operadores básicos aritméticos built-in da linguagem Scala. Abaixo um exemplo de cada um baseado no autor [OMS⁺21].

```
// Criando as variaveis
scala> val n1 = 6
val n1: Int = 6

scala> val n2 = 2
val n2: Int = 2

// Soma
scala> n1 + n2
val res1: Int = 8

// Subtracao
scala> n1 - n2
val res2: Int = 4

// Multiplicacao
scala> n1 * n2
val res3: Int = 12
```



```
// Divisao
scala> n1 / n2
val res4: Int = 3

// Module
scala> n1 % n2
val res6: Int = 0
```

2.3.2 Operadores de comparação

Os operadores são utilizados para comparar valores, sempre retornará valor boolean `true` ou `false`. [OMS⁺21] cita a sintaxe dos operadores abaixo.

- `==` : Igualdade.
- `!=` : Diferença.
- `<` : Menor.
- `>` : Maior.
- `<=` : Menor ou igual.
- `>=` : Maior ou igual.

```
// Declarando variaveis
scala> val a = 3
val a: Int = 3
scala> val b = 8
val b: Int = 8

//Igualdade
scala> a == b
val res7: Boolean = false

// Diferenca
scala> a != b
val res8: Boolean = true

// Menor
scala> a < b
val res10: Boolean = true

//Maior
scala> a > b
val res9: Boolean = false

//Menor ou igual
scala> a <= b
val res11: Boolean = true

//Maior ou igual
scala> a >= b
```

```
val res12: Boolean = false
```

2.3.3 Operadores lógicos

Operadores lógicos usam expressões booleanas (verdadeiro ou falso) para avaliar os valores. [OMS⁺21] afirma que existem 3 tipos

- `&&` : and
- `||` : or
- `!` : not

É necessário saber a tabela verdades dos operadores antes de realizar os códigos. Veremos em baixo cada tabela baseado em [dC11]. Sendo 1 para verdadeiro e 0 para falso.

Tabela 2.1: And

A	B	A And B
0	0	0
0	1	0
1	0	0
1	1	1

Chegamos a conclusão observando a tabela 2.1 que é apenas verdade quando ambos valores de A e B são verdadeiros.

Tabela 2.2: Or

A	B	A Or B
0	0	0
0	1	1
1	0	1
1	1	1

Chegamos a conclusão que observando a 2.2 é verdadeiro quando pelo menos um dos valores de A e B é verdadeiro. Chegamos a conclusão observando a tabela 2.3 que o Not A inverte os

Tabela 2.3: Not

A	Not A
0	1
1	0

valores de A.

Agora sabendo a tabela verdade, iremos demonstrar alguns exemplos em Scala.

```
scala> val a = 5
val a: Int = 5

scala> val b = 8
val b: Int = 8

// ----- AND -----
```

```
scala> a > 0 && b < 10 // Verdadeiro | Verdadeiro
val res2: Boolean = true

a > 10 && b > 10 // Falso | Falso
val res1: Boolean = false

scala> a > 1 && b == 1 // Verdadeiro | Falso
val res4: Boolean = false

//-----OR-----
scala> a == 5 || b != 0 // Verdadeiro | Verdadeiro
val res5: Boolean = true

scala> a < 1 || b == 1 // Falso | Falso
val res6: Boolean = false

scala> a >= 3 || b != 8 // Verdadeiro | Falso
val res7: Boolean = true

//-----NOT-----
scala> !(a > b) // falso -> verdadeiro

scala> !(b != 5) // verdadeiro -> falso
val res9: Boolean = false
```

2.4 Entrada e Saída

Os comandos de entrada e saída são importantes para mostrar mensagens na tela e para receber dados.

2.4.1 Saída

Existem dois comandos para exibir mensagem na tela, citado pelo autor [OMS⁺21], são eles o `println` e o `print`. O `println` adiciona uma quebra de linha no final da exibição

```
scala> println("Hoje e feriado")
Hoje e feriado
```

```
print("Mensagem sem quebra de linha")
Mensagem sem quebra de linha~
```

2.4.2 Entrada

Iremos utilizar a IDE Visual Studio Code para criar nosso script, tendo em vista que no terminal é bom para mostrar pequenas execuções. [Sfr21] explica que para informar dados através do usuários precisamos importar a biblioteca `scala.io.StdIn`. Veremos mais adiante o por que precisamos de uma função `main` para executar nossos programas.

```
import scala.io.StdIn.readLine

def main(args: Array[String]) = {
```

```
        printf("Qual seu nome? ")
// O dado digitado sera armazenado em 'name'
        val name: String = readLine()

// Vai mostrar nome digitado
        printf(s"Seu nome e $name")
// (s"... " permite mostrar o valor de uma variavel com o '$')
    }
```

2.5 Condicionais

Condicionais são estruturas de controle que executa blocos lógicos se a condição for verdadeira. [Sfr21] demonstra que em Scala temos os if, if-else e match

2.5.1 IF

Executa um código apenas se a informação for verdadeira.

```
def main(args: Array[String]) = {
    val x = 10
    if(x > 5)
    {
        println(s"$x e maior que 5")
    }
}
```

Fazemos uma verificação de x, se for maior que 5, irá mostrar uma mensagem. Se for menor, nada acontecerá. Para que uma bloco lógico seja executado se a condição não for verdadeira necessitamos criar uma estrutura if-else.

```
def main(args: Array[String]) = {
    val x = 3
    if(x > 5)
    {
        println(s"$x e maior que 5")
    }
    else{
        printf(s"$x e menor que 5")
    }
}
```

Agora se x for menos que 5, exibiremos uma mensagem na tela.

Há uma forma mais simplificada de fazer o if-else em Scala. Em uma única linha.

```
def main(arg: Array[String]) = {
    val x: Int = 12
    var msg: String = if (x >= 18) "Precisa se
        alistar" else "Nao precisa se alistar"
    println(msg)
}
```

Declaramos uma variável msg que vai armazenar uma mensagem. Se x for maior que 18, vai armazenar a mensagem "Precisa se alistar", caso contrário, "Não precisa se alistar".

2.5.2 match

Para [Sfr21] o match ele permite comparar expressões com uma série de padrões e executar o padrão especificado caso caia na condição. Ele é uma alternativa mais poderosa em comparação ao if-else

```
def main(args: Array[String]) = {  
    val y = 2  
    y match {  
        case 0 => println("Azul")  
        case 1 => print("Vermelho")  
        case 2 => println("Amarelo")  
        case 3 => println("Verde")  
        case _ => println("Cor nao disponivel")  
    }  
    // Vai exibir "Amarelo"  
}
```

Se definirmos y igual a qualquer um dos valores selecionados no comando match que irá exibir o trecho do código. Note que o ultimo case é `_`. Isso significa que se o y for qualquer valor diferente dos definidos anteriormente, ele retornará aquele bloco lógico.

2.6 Estruturas de repetição

Estruturas de repetição são blocos de códigos que se repetem até que uma condição seja verdadeira, ou enquanto uma condição for verdadeira.

2.6.1 while

While pode ser traduzido para "enquanto", [Wam21] define que ,basicamente, ele faz uma execução de código até que a condição definida seja falsa.

```
def main(arg: Array[String]) = {  
    var x = 0  
    while (x <= 5){  
        println(x)  
        x += 1  
    }  
}
```

Enquanto x for menor que 5, iremos executar o bloco de código escrito dentro do while. Internamente, mostramos sempre o valor e atualizamos a variável x para somar +1 a cada interação do while.

2.6.2 For

[Sfr21] argumenta que o for é usado para repetir uma operação sob um finita quantidade de valores, importante para iterar sobre coleções e sequências. Sua estrutura é simples e fácil de entender.

```
def main(args: Array[String]) = {  
    for(i <- 1 to 5)  
    {  
        println("Paradigmas!")  
    }  
}
```

O `i` vai iniciar em 1 e vai até 5, enquanto estiver nesse intervalo irá exibir mensagem. Com o `for` não precisamos fazer `i+=1` porque ele vai atualizar automaticamente.

2.6.3 Foreach

O `foreach` é semelhante ao `for`, porém é mais útil para iterar sobre coleções e sequências. O método `foreach` é usado quando queremos processar cada elemento e executar apenas efeitos colaterais, sem retornar um novo valor. [Wam21]. Iremos mostrar um exemplo com lista, apesar de não termos entrado nesse assunto ainda, mas para um claro entendimento do `foreach` usaremos como exemplo.

```
def main(args: Array[String]) = {  
  val bandas = List("Black Sabbath", "Megadeth",  
    "Nirvana", "Offspring", "Caneta Azul")  
  
  bandas.foreach(nome => println(nome))  
}
```

Primeiro definimos uma lista que possui vários elementos contendo nomes de bandas. Depois usamos o `foreach`. Dizemos em quem ele irá iterar, no caso a variável lista chamada `banda`. Dentro do parênteses, dizemos como vamos nos referir ao conteúdo dentro da lista, nessa caso será de `nome`. E vamos mostrar cada banda (`nome`) até o final do tamanho da lista.

3. Programação em Scala

Este capítulo veremos conceitos mais avançado de Scala.

3.1 Funções

Funções são blocos de códigos que realizam tarefas específicas quando solicitadas. Elas são fundamentais para organizar, reutilizar, abstrair e deixa legível um código. Está presente em quase todas as linguagens. Importante dizer que em Scala todo programa precisa da função main.

O corpo da função segue o seguinte padrão:

```
def main(parametro: tipo): tipo = {}
```

[OMS⁺21] diz que para definir uma função precisamos do comando `def` seguido pela escolha do nome da função. Após isso, nomeamos seus parâmetros e dizemos o tipo de cada um, e por fim, informamos o tipo de retorno da função. Um ponto importante é que nem todas funções recebem parâmetros.

```
def soma(a: Int, b: Int): Int = {  
    return a + b  
}  
  
def main(args: Array[String]) = {  
    val resultado = soma(5,3)  
    printf(resultado)  
}
```

Primeiro criamos a função `soma` que recebe dois números inteiros: `A` e `B`. Essa função retorna um valor inteiro, que é o resultado dos respectivos valores passados na função. Dentro do `main` atribuímos o valor da soma na variável `resultado`. Em seguida, mostramos seu valor.

Um exemplo de função sem parâmetros.

```
def mensagem(): Unit = {  
    println("Esta chovendo hoje")  
}
```

```

    }

    def main(args: Array[String]) = {
        mensagem()
    }

```

Não recebe nenhuma parâmetro e não retorna nada, apenas exibe um `println`, por isso o seu retorno é `Unit` que é semelhante ao `void` em várias outras linguagens.

3.1.1 Função anônima

Segundo [Sfr21], Funções anônimas são funções que você define de forma rápida e concisa. Uma alternativa para as funções padrões de Scala. Vejamos exemplos

```

val soma = {(a: Int, b: Int) => a + b }

soma(1,1)
// OUTPUT: 2

```

Sua criação é feita utilizando `{}`, no qual, inicialmente, declara os parâmetros e, posteriormente, a operação. Em baixo temos a diferença entre a função padrão e a anônima.

```

// normal
def soma(a: Int, b: Int): Int = a + b
// anonima
val soma = {(a: Int, b: Int) => a + b }

```

3.2 Classe

classes são estruturas utilizadas para determinar um objeto e sua interação com o mundo real [Sfr21]. Classes formam a base da programação orientada a objetos. Elas são compostas por métodos e atributos. Iremos demonstrar um exemplo criando uma classe para representar um cachorro.

```

class Cachorro(nome: String, cor: String, idade: Int)
{
    val dogNome: String = nome
    val dogCor: String = cor
    val dogIdade: Int = idade

    def latir(): Unit = {
        println("AU AU")
    }

    def andar(distancia: Float): Unit = {
        println(s"Percorreu $distancia metros")
    }

    def dados(): Unit = {
        println(s"Nome: $dogNome | idade = $dogIdade | cor = $dogCor")
    }
}

```



```

    }

    def main(args: Array[String]) = {
        val bidu = new Cachorro("Bidu", "amarelo", 5)

        bidu.latir() // "AU AU"
        bidu.andar(5) //Percorreu 5 metros
        bidu.dados() // Exibe dados do cachorro
    }

```

Nesse exemplos criamos a classe **Cachorro** com seu nome, cor e idade. Criamos os seus métodos: `latir`: Exibe apenas uma mensagem. `andar` recebe um parâmetro de distancia que exibe a distância percorrida pelo cão; `dados` exibe os dados definidos anteriormente para o cachorro.

Dentro da função `main` instanciamos a classe **Cachorro** para a variável **bidu** que agora tem os métodos definidos pela classe.

3.3 Listas

[Sfr21] afirma que listas são naturalmente implementadas como estrutura de dados encadeada, elas são imutáveis e armazenam somente elementos do mesmo tipo.

```

def main(args: Array[String]) = {
    val pares: List[Int] = List(0, 2, 4, 8)
}

```

Criamos uma variável `par` para armazenar alguns números inteiros pares. O `print` vai exibir todos os valores da lista, mas podemos selecionar pelo índice, sabendo que começa sempre do 0. Vamos demonstrar como funciona esse acesso.

```

def main(args: Array[String]) = {
    val numeros = List(1, 10, 15, 8, 2)
    println(pares(0)) // 1
    println(pares(1)) // 10
    println(pares(2)) // 15
    println(pares(3)) // 8
    println(pares(4)) // 2
}

```

3.3.1 Operações com listas

Vamos demonstrar algumas das operações, baseadas nos exemplos dado pelo autor [OMS⁺21] em seu livro, que podem ser feitas com listas.

```

// Definindo listas
val pares = List(0,2,4,6,8)

val impar = List(1,3,5,7,9)

// Combinar listas
val combinacao = pares ::: impar

```

```
// OUTPUT: (0, 2, 4, 6, 8, 1, 3, 5, 7, 9)

// Conta quantos numeros pares existem
pares.count(n => n % 2 == 0)
//OUTPUT: 5

// Apaga os dois primeiros elementos
pares.drop(2)
// OUTPUT: (4, 6, 8)

//Apaga os dois ultimos elementos
pares.dropRight(2)
// OUTPUT: (0, 2, 4)

pares.exists(s => s > 5)
// OUTPUT: = true

//Mostra apenas valores menores que 5
pares.filter(s=>s < 5)
//OUTPUT:(0, 2, 4)

//Adiciona +10 para cada elemento da lista
pares.map(s => s + 10)
//OUTPUT: (10, 12, 14, 16, 18)

//Inverte a lista
pares.reverse
// OUTPUT: (8, 6, 4, 2, 0)
```

3.4 Array

[OMS⁺21] diz que array são coleções semelhantes aos arrays do Java, armazenam apenas o mesmo tipo, porém seu diferencial é que são mutáveis.

```
val palavras = Array("Diogo", "Costa", "Enzo", "Russo"
    , "Bone")

//Acessa primeiro elemento da tupla
palavra(0)
//Diogo
```

3.5 Vector

[Wam21] comenta que vector armazenam valores imutáveis, porém é implementado como uma árvore de nós, contendo blocos de elementos. Ele tem acesso eficiente para leitura e escrita.

```
val vetor = Vector("Cano", "Andre", "Arias", "Keno")
```

Definimos um vetor que é um Vector de strings que contem 4 elementos. Agora iremos demonstrar alguns métodos e atributos úteis de vector.

```
// Retorna tamanho do vector
vetor.length
// OUTPUT: 5

// Verifica se esta vazio
vetor.isEmpty
// OUTPUT: false

// Retorna o valor no indice especificado
vetor.apply(1)
// OUTPUT: faca

// Retorna novo vetor com valor atualizado
vetor.updated(2, "xicara")
// OUTPUT: (panela, faca, xicara, copo, fogao)

// Retorna vector sem os x primeiros elementos
vetor.drop(1)
// OUTPUT: Vector(faca, garfo, copo, fogao)

// Retorna vector com os x primeiros elementos
vetor.take(3)
//OUTPUT: Vector(panela, faca, garfo)

// Retorna vector em ordem inversa
vetor.reverse
// OUTPUT: (fogao, copo, garfo, faca, panela)
```

3.6 Tuplas

[OMS²¹] ressalta que tuplas são coleções imutáveis que aceitam diferentes tipos de dados. São constantemente utilizadas quando é preciso retornar vários valores de uma função. Sua criação é bem simples.

```
val par = (04, "CCT")
println(par._1) // 04
println(par._2) // CCT
```

Na primeira linha criamos uma tupla com o valor inteiro 04, sendo seu primeiro elemento, em seguida a string "CCT", como seu segundo elemento. Para acessar cada elemento, precisa utilizar `_`. Você pode achar estranho que o primeiro elemento começa em 1, isso se dá ao fato de tradições de outras linguagens que utilizam tuplas estáticas, como Haskell and ML.

Alguns exemplos de métodos de tuplas.

```
val tupla = ("HTML", 5, "CSS", 3)

// Conta quantos elementos tem na tupla

tupla.productArity
//OUTPUT: 4
```

```
//Forma de acessar elementos seguindo principio do
//primeiro elemento ser 0
tupla.productElement(0)
// OUTPUT: HTML

//Copia uma tupla
tupla.copy(_1 = "Scala", _2 = 3) // Mudei o primeiro e
//o segundo elemento
// OUTPUT: (Scala,3,CSS,3)
```

3.7 Set

Segundo [Sfr21] Sets são coleções não ordenadas e sem duplicações. [OMS⁺21] afirma que os Sets podem ser mutáveis ou imutáveis. Por padrão o set é imutável, para mexer com set mutáveis é necessário utilizar a biblioteca `collection.mutable.Set`.

```
// Sets imutaveis
var marcas = Set("Sony", "Xiaomi", "Samsung")
//OUTPUT: (Sony, Xiaomi, Samsung)

println(marcas)
// OUTPUT: (Sony, Xiaomi, Samsung)
```

Set suporta operações e métodos bem úteis

```
// Definindo sets
val p = Set(0, 2, 4)

val q = Set(1, 3, 4)

// Operacao de uniao
p.union(q)
//OUTPUT: (0, 1, 2, 3, 4)

// Operacao de intersecao
p.intersect(q)
//OUTPUT: (4)

// operacao de diferenca
p.diff(q)
//OUTPUT: (0, 2)

// verifica se contem tal valor
p.contains(2)
//OUTPUT: true

// verifica se esta vazio
p.isEmpty
//OUTPUT: false

//Informa o tamanho
```

```
p.size
//OUTPUT: = 3
```

Para operar com sets mutáveis é necessário utilizar biblioteca `scala.collection.mutable.Set`

```
import scala.collection.mutable.Set

var comida = Set("banana", "pudim", "quiche", "amora")

//adicionar novo elemento
comida.add("pipoca")

// Remove um elemento
comida.remove("banana")

println(comida)
// OUTPUT: amora, pudim, pipoca, quiche
```

Importando a biblioteca permite trabalharmos com sets mutáveis. Adicionamos e removemos elementos do set criado, um fator importante é que quando adicionamos um novo valor ao set, não necessariamente ele irá pro final do conjunto. O imutável não aceita as operações de `add` e `remove`, para muda-los é preciso criar um novo set ou atualizar o antigo

```
conj = Set(1, 2, 4)

// Adicionar
conj+= 8 // conj = conj + 8

conj -= 1 // conj = conj - 1

println(conj)
println(2, 4, 8)
```

3.8 Maps

[OMS⁺21] cita que Maps em Scala são estruturas com chaves e valores, cada chave associa a um valor. Elas podem ser mutáveis ou imutáveis.

```
val mapa = Map("suco" -> 13.40f, "vinho" -> 23.30f, "
    chocolate" -> 4.99f, "queijo" -> 8.10f)

mapa("suco")
//OUTPUT 13.4
```

Definimos uma variável `mapa` que recebe um produto e seu respectivo valor. Podemos acessar cada valor informando sua chave. Segue um exemplo para acessar cada chave e valor do map.

```
\\ Exibindo os valores de chave e valor

mapa.foreach{(chave, valor) => println(s"$chave =>
    $valor")}
// OUTPUT:
//suco => 13.4
```

```
//vinho => 23.3
//chocolate => 4.99
//queijo => 8.1
```

Utilizando o `foreach` apresentado no capítulo anterior, passamos dois parâmetros chave e valor, por fim exibimos cada chave e valor. Também seria possível alcançar apenas chave ou valor.

```
// Exibindo apenas as chaves
mapa.keys.foreach{(chaves)=>println(s"$chaves")}
// suco
// vinho
// chocolate
// queijo

// Exibindo apenas os valores
mapa.values.foreach{(valores)=>println(s"$valores")}
// suco
// vinho
// chocolate
// queijo
```

Para trabalhar com Maps mutáveis deve importar `import scala.collection.mutable.Map`

```
import scala.collection.mutable.Map

var moveis = Map("cadeira" -> "quarto", "pia" -> "
    banheiro", "carro" -> "quintal")
```

Vamos mostrar alguns métodos e atributos de maps em Scala.

```
var torneio = Map(1 -> "marcos", 2 -> "alessandro", 3
    -> "sergio")

// Retorna o tamanho do map
torneio.size
//OUTPUT: 3

// Retorna se esta vazio ou nao
torneio.isEmpty
//OUTPUT: false

// Exibe todas as chaves
torneio.keys
//OUTPUT: (1, 2, 3)

// Exibe todos os valores
torneio.values
//OUTPUT: (marcos, alessandro, sergio)

// Retorna o valor da chave X
torneio.get(1)
//OUTPUT: (marcos)
```

```
// Verifica se existe a chave X
torneio.contains("2")
//OUTPUT: true

// Adiciona nova chave e valor ao map
torneio = torneio + ("4" -> "Kevin")
//OUTPUT:(1 -> marcos, 2 -> alessandro, 3 -> sergio, 4
        -> Kevin)

// Remove uma chave
torneio = torneio - "3"
//OUTPUT: (1 -> marcos, 2 -> alessandro, 4 -> Kevin)

// Atualiza uma chave
torneio.updated("1", "larissa")
//OUTPUT: (1 -> larissa, 2 -> alessandro, 4 -> Kevin)
```


4. Aplicações da Linguagem Scala

Nesse capítulo iremos apresentar algumas aplicações da linguagem Scala.

4.1 Calculadora

Vamos construir uma calculadora bem simples que receberá 2 números e realiza as operações mais básicas: soma, subtração, divisão, multiplicação e potenciação.

```
import scala.math

class Calculadora {
    def soma(a: Double, b: Double): Double = a + b

    def subtracao(a: Double, b: Double): Double =
        a - b

    def multiplicacao(a: Double, b: Double):
        Double = a * b

    def divisao(a: Double, b: Double): Double = {
        if (b != 0)
            a / b
        else
        {
            println("ERRO: Nao pode
                dividir por zero")
            return -1
        }
    }
}
```

```
def potencia(a: Double, b: Double): Double =  
    math.pow(a,b)  
}  
  
def main(args:Array[String]) = {  
    val a = 8  
    val b = 2  
    val calc = new Calculadora()  
  
    val soma = calc.soma(a, b)  
    val sub = calc.subtracao(a, b)  
    val mult = calc.multiplicacao(a, b)  
    val div = calc.divisao(a, b)  
    val pot = calc.potencia(a, b)  
  
    println(s"A soma de $a + $b = $soma ")  
    println(s"A subtracao de $a - $b = $sub ")  
    println(s"A multiplicacao de $a * $b = $mult "  
    )  
    println(s"A divisao de $a / $b = $div ")  
    println(s"A exponenciacao de $a ^ $b = $pot ")  
}
```

Foi criada uma classe `Calculadora` que possui cinco métodos, onde cada um recebe dois parâmetros do tipo `double`. No método da divisão, há a verificação para saber se `b` é menor que 0, caso seja, exibe uma mensagem de erro e retorna -1. No `main` são realizados os cálculos, iniciando a classe calculadora como `calc`, e são feitas as operações.

Figura 4.1: Código Fonte

```
1  import scala.math
2
3  class Calculadora {
4      def soma(a: Double, b: Double): Double = a + b
5
6      def subtracao(a: Double, b: Double): Double = a - b
7
8      def multiplicacao(a: Double, b: Double): Double = a * b
9
10     def divisao(a: Double, b: Double): Double = {
11         if (b != 0)
12             a / b
13         else
14             {
15                 println("ERRO: Nao pode dividir por zero")
16                 return -1
17             }
18     }
19
20     def potencia(a: Double, b: Double): Double = math.pow(a,b)
21 }
22
23 def main(args:Array[String]) = {
24     val a = 8
25     val b = 2
26     val calc = new Calculadora()
27
28     val soma = calc.soma(a, b)
29     val sub = calc.subtracao(a, b)
30     val mult = calc.multiplicacao(a, b)
31     val div = calc.divisao(a, b)
32     val pot = calc.potencia(a, b)
33
34     println(s"A soma de $a + $b = $soma ")
35     println(s"A subtracao de $a - $b = $sub ")
36     println(s"A multiplicacao de $a * $b = $mult ")
37     println(s"A divisao de $a / $b = $div ")
38     println(s"A exponenciacao de $a ^ $b = $pot ")
39 }
```

Figura 4.2: Output

```
A soma de 8 + 2 = 10.0
A subtracao de 8 - 2 = 6.0
A multiplicacao de 8 * 2 = 16.0
A divisao de 8 / 2 = 4.0
A exponenciacao de 8 ^ 2 = 64.0
```

4.2 Cilindro

Nesse código iremos fazer uma demonstração de calculo da área e volume de um cilindro qualquer.

```
import scala.math
import scala.io.StdIn

class Cilindro{

    def volume(raio: Double, altura: Double) : Double = (
        math.Pi * raio * raio) * altura

    def area(raio: Double, altura: Double): Double = (2 *
        math.Pi * raio) * (raio + altura)
}

def main(args:Array[String]) = {

    var done = true

    val cilindro = new Cilindro()

    while (done) {
        println("[1]-Inserir dados\n[2]-Sair")
        val opcao = StdIn.readInt()

        if (opcao == 1) {
            println("Digite o raio do cilindro:")
            val r = StdIn.readDouble()

            println("Digite a altura do cilindro:")
            val h = StdIn.readDouble()

            val volume = cilindro.volume(r, h)
            val area = cilindro.area(r, h)

            println(s"O volume do cilindro e ${"%.2f".format(volume)} e sua area e ${"%.2f".format(area)}")

        }
        else if (opcao == 2) {
            done = false
            println("Encerrando...")
        }
        else {

```

```

        println("ERRO: Digite apenas opcoes
                viaveis.")
    }

    println()
}
}

```

Foi criado um classe Cilindro com dois métodos volume e area que recebem como parâmetro dois tipos Double e retorna double.

No main, foi criado um programa interativo em loop até que o usuário digite a opção 2 para encerrar. Foram criadas variáveis r e h para receber, respectivamente, raio e altura; com isso foram feitos os cálculos e exibidos o volume e área.

Código fonte feito na IDE Visual Studio Code

Figura 4.3: Código Fonte

```

1  import scala.math
2  import scala.io.StdIn
3
4
5  class Cilindro{
6
7      def volume(raio: Double, altura: Double) : Double = (math.Pi * raio * raio) * altura
8
9      def area(raio: Double, altura: Double): Double = (2 * math.Pi * raio) * (raio + altura)
10
11  }
12
13  def main(args:Array[String]) = {
14
15      var done = true
16
17      val cilindro = new Cilindro()
18
19
20      while (done) {
21          println("[1]-Inserir dados\n[2]-Sair")
22          val opcao = StdIn.readInt()
23
24          if (opcao == 1) {
25              println("Digite o raio do cilindro:")
26              val r = StdIn.readDouble()
27
28              println("Digite a altura do cilindro:")
29              val h = StdIn.readDouble()
30
31              val volume = cilindro.volume(r, h)
32              val area = cilindro.area(r, h)
33
34              println(s"O volume do cilindro e ${"%0.2f".format(volume)} e sua area e ${"%0.2f".format(area)}")
35
36          }
37      }
38  }

```

Figura 4.4: Autor

Figura 4.5: Código Fonte Visual Studio Code

```
37     else if (opcao == 2) {  
38         done = false  
39         println("Encerrando...")  
40     }  
41     else {  
42         println("ERRO: Digite apenas opcoes viaveis.")  
43     }  
44  
45     println()  
46 }  
47 }  
48
```

Figura 4.6: output

```
[1]-Inserir dados  
[2]-Sair  
1  
Digite o raio do cilindro:  
10  
Digite a altura do cilindro:  
5  
O volume do cilindro e 1570,80 e sua area e 942,48  
  
[1]-Inserir dados  
[2]-Sair  
1  
Digite o raio do cilindro:  
9  
Digite a altura do cilindro:  
8  
O volume do cilindro e 2035,75 e sua area e 961,33  
  
[1]-Inserir dados  
[2]-Sair  
2  
Encerrando...
```

4.3 Parametrização de Retas

Nesse código iremos transformar dois pontos no espaço em uma equação paramétrica.

```
class Reta(val ponto1: (Double, Double), val ponto2: (
    Double, Double)) {

    def calcularEquacaoParametrica(t: Double): String =
    {
        val vetor = (ponto2._1 - ponto1._1, ponto2._2
            - ponto1._2)
        val x = ponto1._1 + vetor._1 * t
        val y = ponto1._2 + vetor._2 * t
        s"\n${ponto1._1} + (${vetor._1})t = ($x) \n${
            ponto1._2} + (${vetor._2})t = ($y)"
    }

    def main(args: Array[String]) = {
        val reta = new Reta((1, 0), (2, 4))
        val pontoParametrico = reta.
            calcularEquacaoParametrica(0.5)
        println(s"Escrevendo na equacao parametrica da
            reta: $pontoParametrico")
    }
}
```

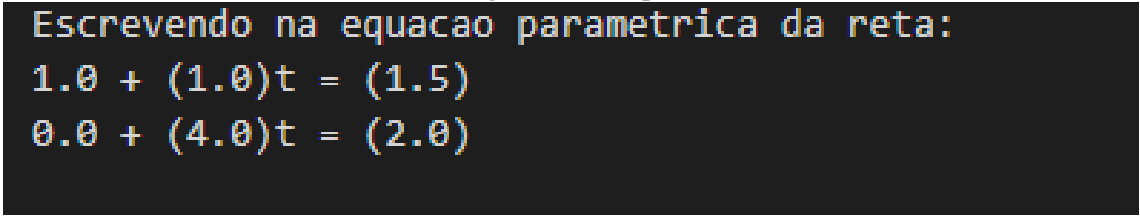
O código cria uma classe "Reta" que calcula e exibe a equação paramétrica da reta dado por dois pontos. No main, é criada a instância da reta e definidos os pontos (1,0) e (2,4), com isso são feitos os cálculos com o método `calcularEquacaoParametrica` e passando o valor de `t` como argumento.

Código fonte feito na IDE Visual Studio Code

Figura 4.7: Código fonte

```
1 class Reta(val ponto1: (Double, Double), val ponto2: (Double, Double)) {
2
3     def calcularEquacaoParametrica(t: Double): String = {
4         val vetor = (ponto2._1 - ponto1._1, ponto2._2 - ponto1._2)
5         val x = ponto1._1 + vetor._1 * t
6         val y = ponto1._2 + vetor._2 * t
7         s"\n${ponto1._1} + (${vetor._1})t = ($x) \n${ponto1._2} + (${vetor._2})t = ($y)"
8     }
9 }
10
11 // Exemplo de uso:
12
13 def main(args: Array[String]) = {
14     val reta = new Reta((1, 0), (2, 4))
15     val pontoParametrico = reta.calcularEquacaoParametrica(0.5)
16     println(s"Escrevendo na equacao parametrica da reta: $pontoParametrico")
17 }
18
```

Figura 4.8: Output



```
Escrevendo na equacao parametrica da reta:  
1.0 + (1.0)t = (1.5)  
0.0 + (4.0)t = (2.0)
```


4.4 SelectionSort

Nosso próximo código é o SelectionSort, um algoritmo básico de ordenação que funciona pegando o menor elemento do array e o colocando na posição correta Array. Em nosso exemplo iremos utilizar um array de 50 elementos aleatórios.

```
import scala.util.Random

class SelectionSort {
  def selectionSort(array: Array[Int]): Unit = {
    val size = array.length
    for (i <- 0 until size - 1) {
      var minIndex = i
      for (j <- i + 1 until size) {
        if (array(j) < array(minIndex)) {
          minIndex = j
        }
      }
      if (minIndex != i) {
        val temp = array(i)
        array(i) = array(minIndex)
        array(minIndex) = temp
      }
    }
  }
}

object Main {
  def randomValues(array: Array[Int]): Unit = {
    val random = new Random()
    for (i <- 0 until array.length - 1) {
      array(i) = random.nextInt(100)
    }
  }

  def main(args: Array[String]): Unit = {
    val array = new Array[Int](20)
    randomValues(array)

    println("Array original: " + array.mkString(", "))

    val sort = new SelectionSort()
    sort.selectionSort(array)

    println("Array ordenado: " + array.mkString(", "))
  }
}
```

Foi criada uma classe `SelectionSort` e possui o método `selectionSort` que recebe um array de inteiros. Ele ordena percorrendo todo o array, e buscando o menor elemento de cada iteração, e com isso, o coloca na posição correta.

No main, é criado `randomValues` que preenche o array com valores aleatórios; em seguida, no main é definido um array de tipo inteiro com tamanho pré-definido de 20. É exibido o array original, e em seguida foi criada uma instância da classe `SelectionSort` e realiza a ordenação do array. No final, é exibido o array ordenado.

Código fonte feito na IDE Visual Studio Code

Figura 4.9: Código Fonte

```
1  import scala.util.Random
2
3  class SelectionSort {
4    def selectionSort(array: Array[Int]): Unit = {
5      val size = array.length
6      for (i <- 0 until size - 1) {
7        var minIndex = i
8        for (j <- i + 1 until size) {
9          if (array(j) < array(minIndex)) {
10             minIndex = j
11          }
12        }
13        if (minIndex != i) {
14          val temp = array(i)
15          array(i) = array(minIndex)
16          array(minIndex) = temp
17        }
18      }
19    }
20  }
21
22  object Main {
23    def randomValues(array: Array[Int]): Unit = {
24      val random = new Random()
25      for (i <- 0 until array.length - 1) {
26        array(i) = random.nextInt(100)
27      }
28    }
29
30    def main(args: Array[String]): Unit = {
31      val array = new Array[Int](20)
32      randomValues(array)
33
34      println("Array original: " + array.mkString(", "))
```

Figura 4.10: Código Fonte

```
35
36    val sort = new SelectionSort()
37    sort.selectionSort(array)
38
39    println("Array ordenado: " + array.mkString(", "))
40  }
41 }
42
```

Figura 4.11: Autor

```
PS C:\Almeida\Estudos\Scala\estudos\aplicacoes> scala selectionSort.scala
Array original: 47, 29, 83, 55, 3, 32, 69, 69, 41, 40, 94, 35, 29, 35, 29, 35, 41, 97, 80, 0

Array ordenado: 0, 3, 29, 29, 29, 32, 35, 35, 35, 40, 41, 41, 47, 55, 69, 69, 80, 83, 94, 97
PS C:\Almeida\Estudos\Scala\estudos\aplicacoes> █
```

4.5 QuickSort

] Um algoritmo mais sofisticado de ordenação é QuickSort. Nele é selecionado um elemento como pivô, que é utilizado para encontrar a posição correta no array. Os elementos menores ficam a esquerda, enquanto que os maiores ficam a direitas. Após isso, é realizado recursão para organizar o array e combinar ambas partes ordenadas.

```
import scala.util.Random

object QuickSort {
  def randomValues(array: Array[Int]): Unit = {
    val random = new Random()
    for (i <- 0 until array.length) {
      array(i) = random.nextInt(150)
    }
  }

  def quickSort(array: Array[Int]): Unit = {
    def splitArray(low: Int, high: Int): Int = {
      val pivot = array(high)
      var i = low

      for (j <- low until high) {
        if (array(j) < pivot) {
          val temp = array(i)
          array(i) = array(j)
          array(j) = temp
          i += 1
        }
      }

      val temp = array(i)
      array(i) = array(high)
      array(high) = temp
      i
    }

    def sort(low: Int, high: Int): Unit = {
      if (low < high) {
        val pivotIndex = splitArray(
          low, high)
        sort(low, pivotIndex - 1)
        sort(pivotIndex + 1, high)
      }
    }

    sort(0, array.length - 1)
  }

  def main(args: Array[String]): Unit = {
    val array = new Array[Int](20)
```

```

        randomValues(array)

        println("Array original: " + array.mkString(",
            "))

        quickSort(array)
        println("\nArray ordenado: " + array.mkString(
            ", "))
    }
}

```

O quickSort é algoritmo que recebe um array de inteiros e realiza sua ordenação utilizando a estratégia de divisão e conquista. Encontra-se o pivô, e o coloca todos elementos menores que ele a esquerda, e os maiores a direita na função splitArray. Posteriormente, o método sort é chamado para ordenar as duas divisões geradas por recursividade.

É criado um array de 20 elementos que é preenchido aleatoriamente pela método randomValues. Ordena os valores chamando o método quickSort e exibe o array antes e depois.

Código fonte feito na IDE Visual Studio Code

Figura 4.12: Código fonte

```

1  import scala.util.Random
2
3  object QuickSort {
4      def randomValues(array: Array[Int]): Unit = {
5          val random = new Random()
6          for (i <- 0 until array.length) {
7              array(i) = random.nextInt(150)
8          }
9      }
10
11     def quickSort(array: Array[Int]): Unit = {
12         def splitArray(low: Int, high: Int): Int = {
13             val pivot = array(high)
14             var i = low
15
16             for (j <- low until high) {
17                 if (array(j) < pivot) {
18                     val temp = array(i)
19                     array(i) = array(j)
20                     array(j) = temp
21                     i += 1
22                 }
23             }
24
25             val temp = array(i)
26             array(i) = array(high)
27             array(high) = temp
28             i
29         }
30     }

```

Figura 4.13: Código fonte

```
31  def sort(low: Int, high: Int): Unit = {  
32    if (low < high) {  
33      val pivotIndex = splitArray(low, high)  
34      sort(low, pivotIndex - 1)  
35      sort(pivotIndex + 1, high)  
36    }  
37  }  
38  
39  sort(0, array.length - 1)  
40 }  
41  
42  def main(args: Array[String]): Unit = {  
43    val array = new Array[Int](20)  
44  
45    randomValues(array)  
46  
47    println("Array original: " + array.mkString(", "))  
48  
49    quickSort(array)  
50    println("\nArray ordenado: " + array.mkString(", "))  
51  }  
52 }  
53
```

Figura 4.14: Output

```
PS C:\Almeida\Estudos\Scala\estudos\aplicacoes> scala quickSort.scala  
Array original: 90, 41, 87, 40, 140, 75, 69, 22, 44, 68, 77, 42, 77, 149, 69, 51, 122, 57, 9, 66  
  
Array ordenado: 9, 22, 40, 41, 42, 44, 51, 57, 66, 68, 69, 69, 75, 77, 77, 87, 90, 122, 140, 149  
PS C:\Almeida\Estudos\Scala\estudos\aplicacoes> 
```

4.6 Soma de Matrizes

A próxima demonstração é uma soma dos elementos de uma matriz. O código gera uma matriz 3x3 preenchida com valores aleatórios. É criada uma função para exibir a matriz e calcular sua soma.

```
import scala.util.Random

def somaMatriz(matriz: Array[Array[Int]]): Int = {

    var sum = 0

    for(i <- 0 until matriz.length)
    {
        for(j <- 0 until matriz(i).length)
        {
            sum = sum + matriz(i)(j);
        }
    }

    return sum
}

def randomValues(matriz: Array[Array[Int]]): Unit = {
    val random = new Random()

    for (i <- 0 until matriz.length) {
        for (j <- 0 until matriz(i).length) {
            matriz(i)(j) = random.nextInt(100)
        }
    }
}

def printMatriz(matriz: Array[Array[Int]]): Unit = {

    for (i <- 0 until matriz.length) {
        for(j <- 0 until matriz(i).length)
        {
            print(s"${matriz(i)(j)} ")
        }
        println()
    }
}

def main(args: Array[String]) = {

    val matriz = Array.ofDim[Int](3, 3)

    randomValues(matriz)
```

```
println("A matriz gerada e: ")

printMatriz(matriz)

val somaElem = somaMatriz(matriz)
println(s"Soma dos elementos da matriz:
    $somaElem")
}
```

É criada a função `somaMatriz` a qual calcula a soma de todos os elementos da matriz. É criado uma função para preencher a matriz com valores aleatórios, e uma para exibir a matriz. No main, é criado matriz 3x3, preenchida e exibida a matriz e sua soma.


```
1  import scala.util.Random
2
3
4  def somaMatriz(matriz: Array[Array[Int]]): Int = {
5
6      var sum = 0
7
8      for(i <- 0 until matriz.length)
9      {
10         for(j <- 0 until matriz(i).length)
11         {
12             sum = sum + matriz(i)(j);
13         }
14     }
15     return sum
16 }
17
18
19 def randomValues(matriz: Array[Array[Int]]): Unit = {
20     val random = new Random()
21
22     for (i <- 0 until matriz.length) {
23         for (j <- 0 until matriz(i).length) {
24             matriz(i)(j) = random.nextInt(100)
25         }
26     }
27 }
28
29 def printMatriz(matriz: Array[Array[Int]]): Unit = {
30
31     for (i <- 0 until matriz.length) {
32         for(j <- 0 until matriz(i).length)
33         {
34             print(s"${matriz(i)(j)} ")
35         }
36         println()
37     }
38 }
```

Figura 4.15: Código Fonte

```
40  v def main(args: Array[String]) = {  
41  
42      val matriz = Array.ofDim[Int](3, 3)  
43  
44      randomValues(matriz)  
45  
46      println("A matriz gerada e: ")  
47  
48      printMatriz(matriz)  
49  
50      val somaElem = somaMatriz(matriz)  
51      println(s"Soma dos elementos da matriz: $somaElem")  
52  }
```

Figura 4.16: Código Fonte

```
[Running] scala "c:\Almeida\Estudos\Scala\estudos\aplicacoes\somaMatrizes.scala"  
A matriz gerada e:  
61 82 55  
44 5 8  
16 90 35  
Soma dos elementos da matriz: 396
```

Figura 4.17: output

4.7 Aplicações Biologia

Veremos um exemplo simples de uma aplicação biológica para cálculo de IMC. O cálculo é feito com base no peso dividido pelo quadrado da altura.

```
import scala.math

def calculoIMC(peso: Double, altura: Double): Double =
{
    return peso / math.pow(altura, 2)
}

def grauIMC(imc: Double): Unit =
{
    if(imc <= 16.9)
    {
        println("Muito abaixo do peso")
    }

    else if (imc <= 18.4)
    {
        println("Abaixo do peso")
    }

    else if (imc <= 24.9)
    {
        println("Peso normal")
    }

    else if (imc <= 29.9)
    {
        println("Acima do peso")
    }

    else if (imc <= 34.9)
    {
        println("Obesidade I")
    }

    else if (imc <= 40)
    {
        println("Obesidade II")
    }

    else
    {
        println("Obesidade III")
    }
}
```

```
def main(args: Array[String]) = {  
  
    val imc = calculoIMC(58, 1.65)  
  
    println("-----")  
    println(s"Seu IMC e: ${"%0.2f".format(imc)}")  
    print(s"Sua classificacao e: ")  
    grauIMC(imc)  
    println("-----")  
}
```

Foi criado uma função para calcular IMC que recebe dois parâmetros Double e retorna Double. Ainda foi feito uma função grauIMC para informar qual grau da pessoa. No main foi passado valores para a função e armazenado seu retorno na variável imc. Após isso, foi exibido o IMC e seu grau.

Figura 4.18: Código fonte

```
1  import scala.math
2
3  def calculoIMC(peso: Double, altura: Double): Double =
4  {
5      |   return peso / math.pow(altura, 2)
6  }
7
8  def grauIMC(imc: Double): Unit =
9  {
10     |   if(imc <= 16.9)
11     |   {
12     |       |   println("Muito abaixo do peso")
13     |   }
14
15     |   else if (imc <= 18.4)
16     |   {
17     |       |   println("Abaixo do peso")
18     |   }
19
20     |   else if (imc <= 24.9)
21     |   {
22     |       |   println("Peso normal")
23     |   }
24
25     |   else if (imc <= 29.9)
26     |   {
27     |       |   println("Acima do peso")
28     |   }
29
30     |   else if (imc <= 34.9)
31     |   {
32     |       |   println("Obesidade I")
33     |   }
34
35     |   else if (imc <= 40)
36     |   {
37     |       |   println("Obesidade II")
38     |   }
39 }
```

Figura 4.19: Código fonte

```
40
41     else
42     {
43         println("Obesidade III")
44     }
45 }
46
47
48 def main(args: Array[String]) = {
49
50     val imc = calculoIMC(58, 1.65)
51
52     println("-----")
53     println(s"Seu IMC e: ${"%0.2f".format(imc)}")
54     print(s"Sua classificacao e: ")
55     grauIMC(imc)
56     println("-----")
57 }
```

Figura 4.20: Output

```
PS C:\Almeida\Estudos\Scala\estudos\aplicacoes> scala IMC.scala
-----
Seu IMC e: 21,30
Sua classificacao e: Peso normal
-----
```

4.8 Aplicação física

Na aplicação de física vamos demonstrar o cálculo de potência elétrica.

```
import scala.io.StdIn

def calcularPotencia(tensao: Double, corrente: Double)
  : Double =
{
    return tensao * corrente
}

def main(args: Array[String]) =
{
    println("tensao: ")
    val tensao = StdIn.readDouble()
    println("corrente: ")
    val corrente = StdIn.readDouble()

    var potencia = calcularPotencia(tensao,
        corrente)

    println(s"A potencia e: ${"%0.2f".format(
        potencia)}A")
}
```

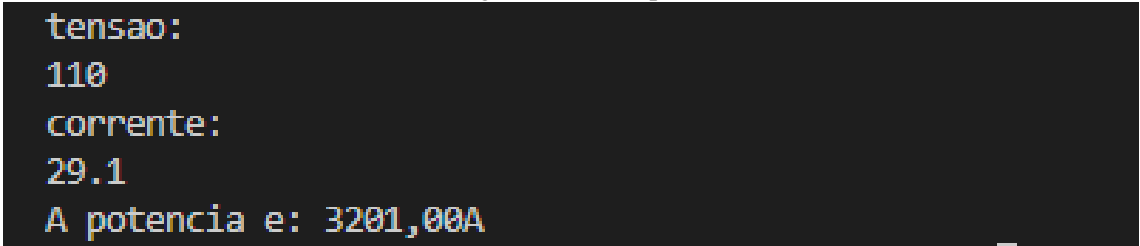
Foi criado a função `calcularPotencia` que recebe dois parâmetros do tipo `Double` e retorna também tipo `double`. A função calcula a potência a partir da tensão e corrente. No `main` é pedido ao usuário para informar os valores para tensão e corrente. No final, é realizado o cálculo da potência e exibido o valor no console.

Código fonte feito na IDE Visual Studio Code

Figura 4.21: Código Fonte

```
1  import scala.io.StdIn
2
3  def calcularPotencia(tensao: Double, corrente: Double): Double =
4  {
5      return tensao * corrente
6  }
7
8  def main(args: Array[String]) =
9  {
10     println("tensao: ")
11     val tensao = StdIn.readDouble()
12     println("corrente: ")
13     val corrente = StdIn.readDouble()
14
15     var potencia = calcularPotencia(tensao, corrente)
16
17     println(s"A potencia e: ${"%0.2f".format(potencia)}A")
18 }
```

Figura 4.22: Output



```
tensao:  
110  
corrente:  
29.1  
A potencia e: 3201,00A
```

The image shows a terminal window with a black background and light blue text. It displays the output of a Scala program. The first two lines are 'tensao:' followed by '110'. The next two lines are 'corrente:' followed by '29.1'. The final line is 'A potencia e: 3201,00A'. There is a small white cursor icon at the end of the last line.

5. Ferramentas existentes e utilizadas

Nesse capítulo serão apresentados duas ferramentas que foram consultadas e utilizadas para realizar o trabalho.

5.1 Visual Studio Code

O Visual Studio Code (VSCode) é um editor de código independente de plataforma desenvolvido pela Microsoft e lançado sob uma licença proprietária. Ele contém apenas pequenas modificações da base de código livre e de código aberto Code - OSS. [Pla20]. O visual Studio Code suporta diversas linguagens, tais como Scala, Python, C, JavaScript e etc. Possui uma interface bem simples e intuitiva.

[Pla20] diz que VSCode começou como um projeto em Mônaco em 2011 com o objetivo de criar um editor que pudesse rodar no navegador. Com a contratação de Erich Gamma como líder do projeto, conhecido por estar envolvido no desenvolvimento de Eclipse. Posteriormente, decidiram mudar o projeto para ser um programa de desktop. Foi anunciado ao público em 2015, e lançado de fato em Abril de 2016.

Seu diferencial são suas extensões que permitem adicionar novas funcionalidades, tornando o ambiente personalizado. Em nosso livro, foram utilizado: `Scala Syntax (official)` e `Scala (Metals)` para rodar a linguagem Scala, e o `Code runner` para compilar os programas.

Seu site é: <https://code.visualstudio.com/>, foi utilizada a versão 1.79.1

Aqui está umas imagens da ferramenta

Figura 5.1: Visual Studio Code

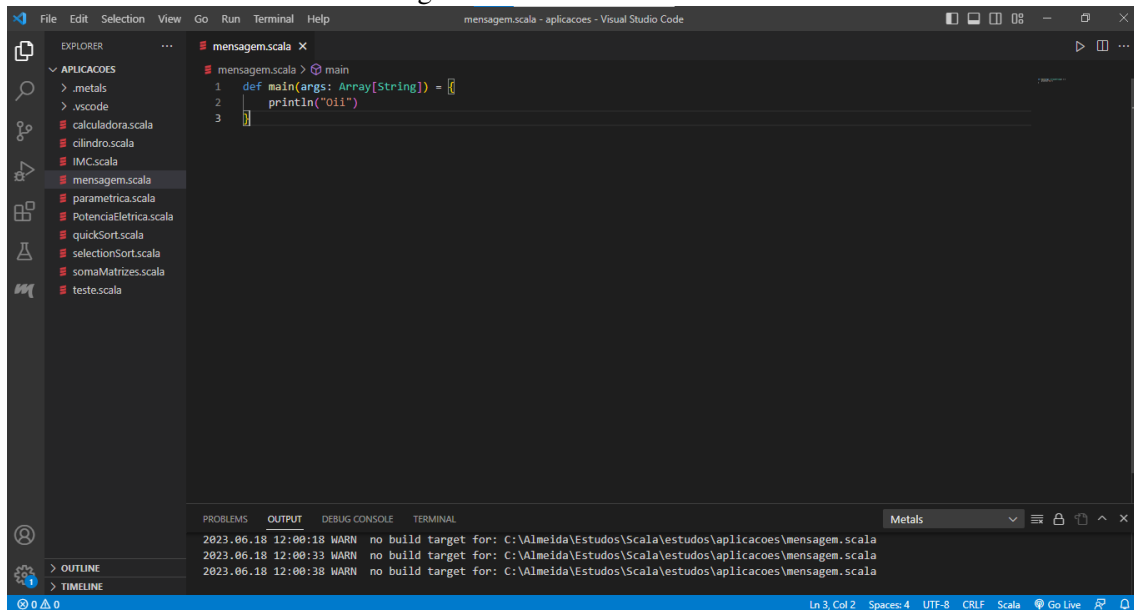


Figura 5.2: Visual Studio Code

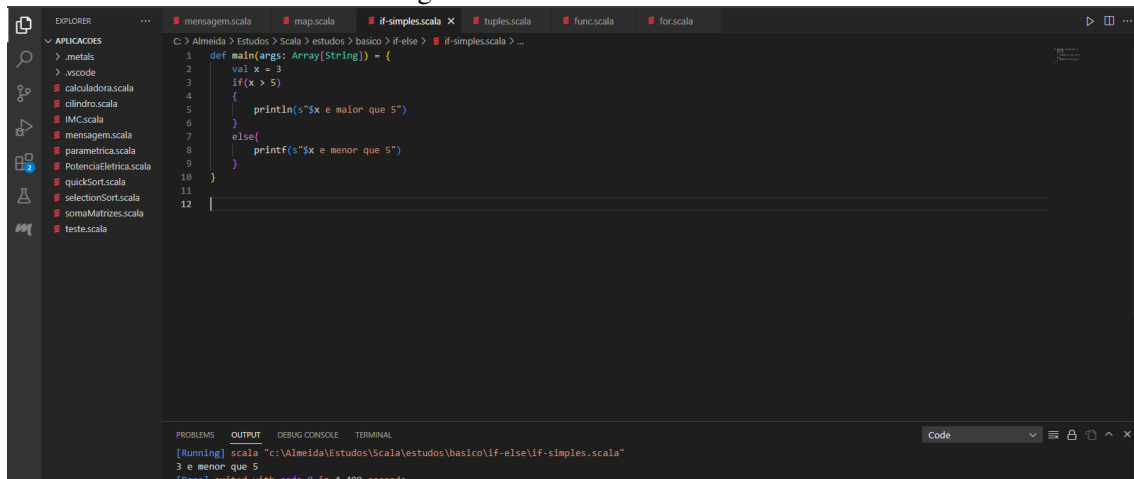


Figura 5.4: Visual Studio Code

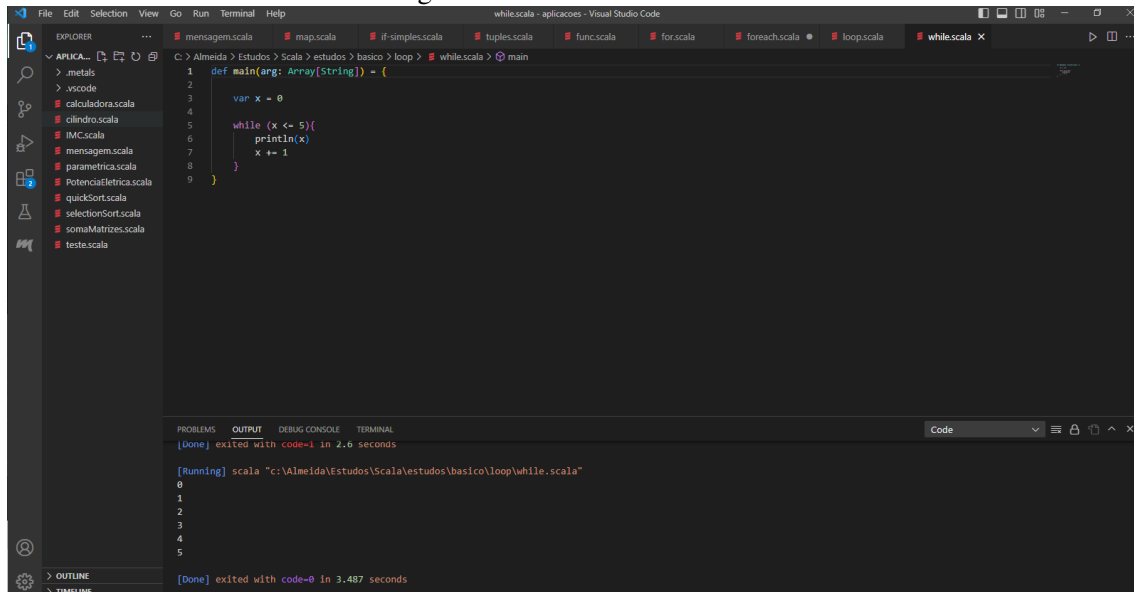
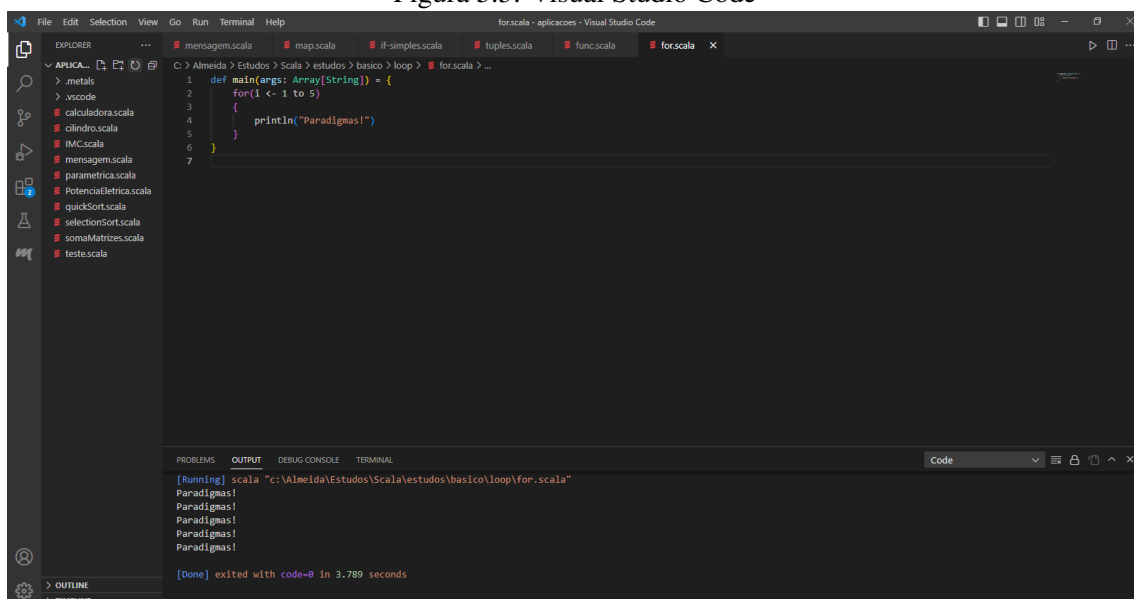


Figura 5.3: Visual Studio Code



5.2 Scastie

O Scastie é um compilador online para Scala disponível no próprio site da linguagem. Foi criado para ser uma ferramenta de fácil acesso, sua interface é simples e intuitiva, não precisa instalar nenhum software na máquina, sendo uma das suas grandes vantagens.

Outro ponto importante do Scastie, é possível compartilhar o link do código com outras pessoas, facilitando a interação com outros programadores. Além disso, o Scastie auxilia na implementação do código, dando sugestões de autocomplemento.

Pode ser acessado em <https://scastie.scala-lang.org>

Aqui estão imagens da ferramenta

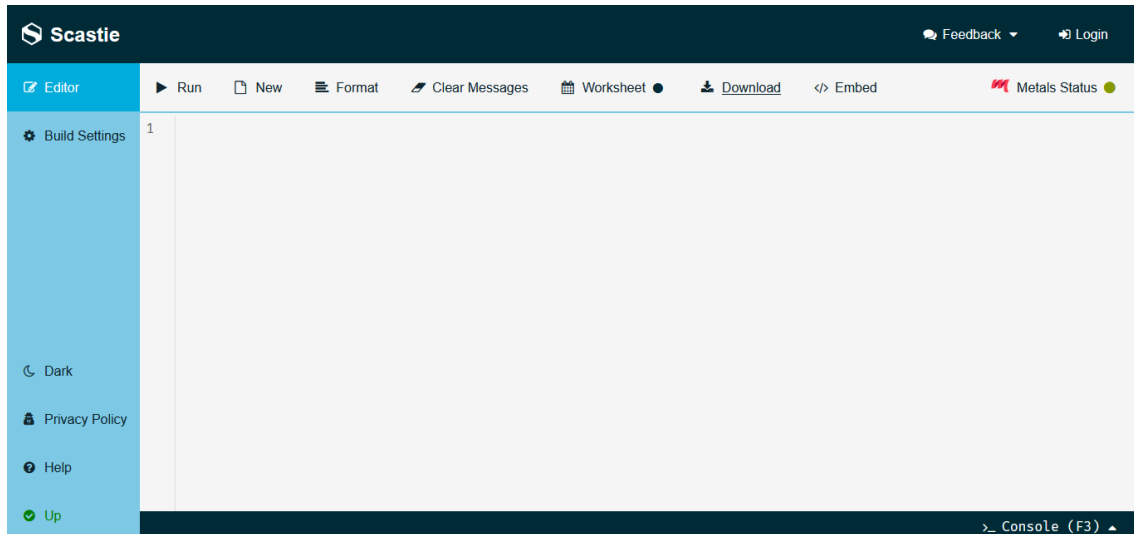


Figura 5.5:



Figura 5.6:

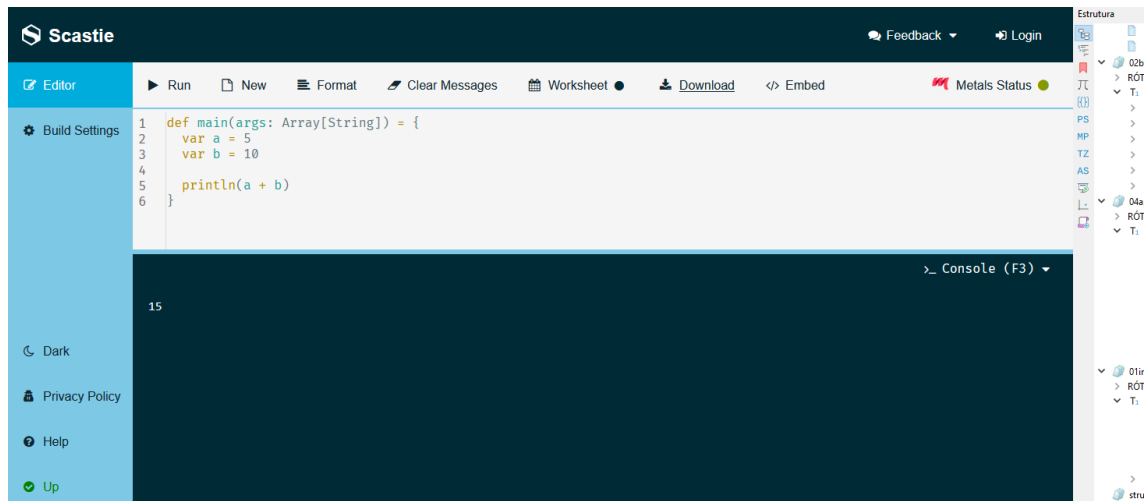


Figura 5.7:



6. Considerações Finais

Neste livro introdutório a respeito da linguagem Scala, foi desenvolvida a história da linguagem, passando pelas suas origens, atuação no mercado de trabalho. Foi passado todo possível potencial da Scala para os anos seguintes

Foram citados suas vantagens de utilização com a JVM, o qual se destaca em relação a outras linguagens, sua combinação de paradigmas orientado a objetos e funcional.

Além disso, foram explorados todos os seus conceitos básicos, como os tipos de dados, estruturas de controle e repetição, operadores e expressões; Assim como seus conceitos mais avançados, como funções, classes e tipos de dados compostos. Além disso, foram demonstradas aplicações práticas da linguagem com diversos exemplos comentados passo a passo.

O livro foi desenvolvido para ser o primeiro contato com a linguagem, portanto, não foram abordados diversos conteúdos que poderiam ser estudados, por exemplo conceitos mais avançados, como: aprofundamento no paradigma orientado a objetos e funcional, **Pattern Matching**, **programação assíncrona** e os frameworks, tais como os citados no livro: **Akka**, **Spark**, **ScalaNLP**, e outras que não foram citadas no livro: **Slick**, **Finagle** e **Spray**.

A seguir, estão a recomendação de alguns livros para se aprofundar na Scala.

Figura 6.1: Linguagens de programação modernas e um bom livro



Figura 6.2: Livros indicados

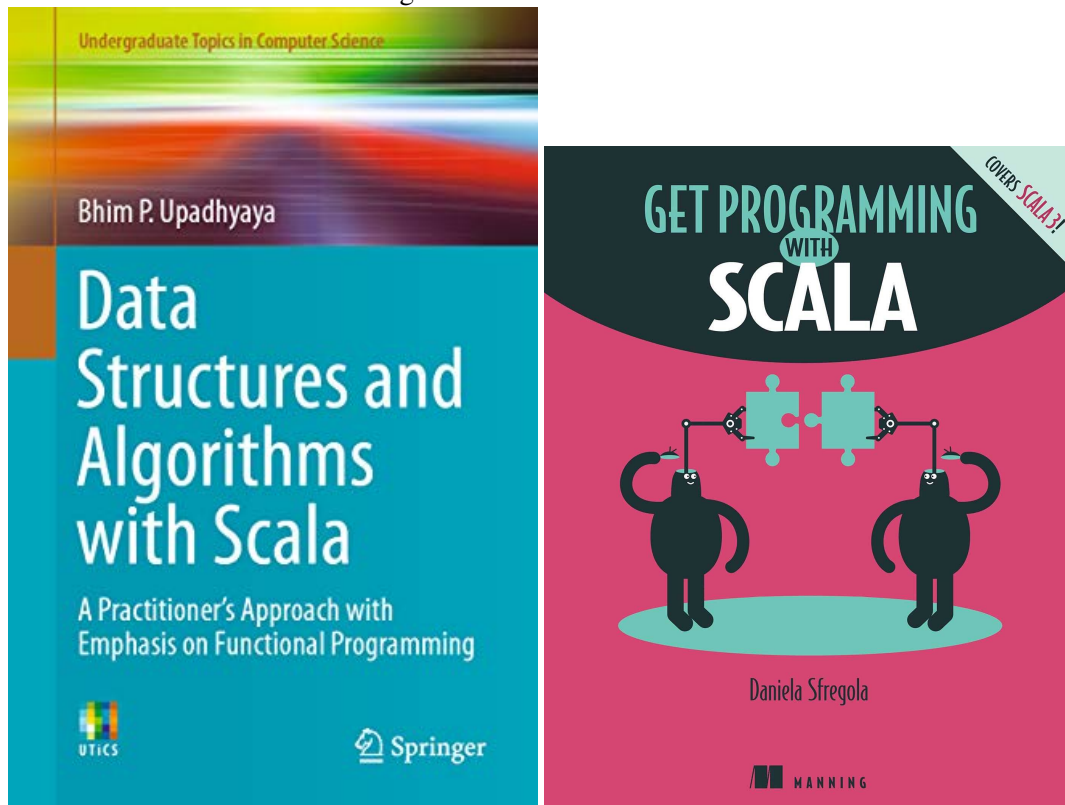
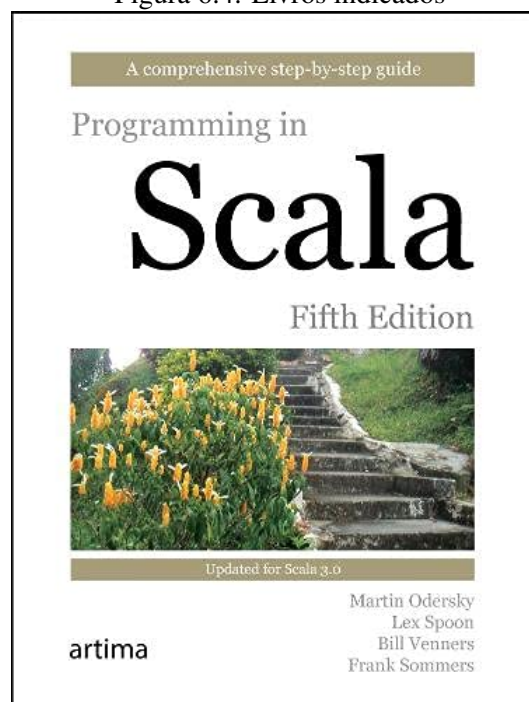


Figura 6.3:

Figura 6.4: Livros indicados





Referências Bibliográficas

- [dC11] Gilles de Castro. Tabelas-verdade (com aplicações para operações em conjuntos). 2011. Citado na página 18.
- [OMea06] Odersky, Martin, and et al. An overview of the scala programming language. Technical report, École Polytechnique Fédérale de Lausanne (EPFL), 2006. Citado na página 8.
- [OMS⁺21] Odersky, Martin, Spoon, Lex, Venners, Bill, Sommers, and Frank. *Programming in Scala Fifth Edition*. Artima Press, Walnut Creek, CA, 5 edition, June 2021. Citado 13 vezes nas páginas 8, 14, 15, 16, 17, 18, 19, 23, 25, 26, 27, 28 e 29.
- [Pla20] Michael Plainer. Study of visual studio code. Technical report, Technische Universität München, 2020. Citado na página 57.
- [Sfr21] Daniela Sfregola. *Get Programming with Scala*. Manning Publications Co. LLC, Shelter Island, NY, 2021. Citado 8 vezes nas páginas 8, 9, 19, 20, 21, 24, 25 e 28.
- [VS09] Bill Venners and Frank Sommers. The origins of scala a conversation with martin odersky, part i. 2009. Citado 2 vezes nas páginas 7 e 8.
- [Wam21] Dean Wampler. *Programming Scala Scalability = Functional Programming + Objects*. O'Reilly Media, Incorporated, Sebastopol, CA, 2021. Citado 7 vezes nas páginas 13, 14, 15, 16, 21, 22 e 26.

