



Introdução à Linguagem Scala

Paradigmas de Linguagens de Programação

**Gabriel Costa Fassarella
Ausberto S. Castro Vera**

22 de junho de 2023



Disciplina: *Paradigmas de Linguagens de Programação 2023*

Linguagem: *Scala*

Aluno: *Gabriel Costa Fassarella*

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)	
	Nota Final do trabalho:

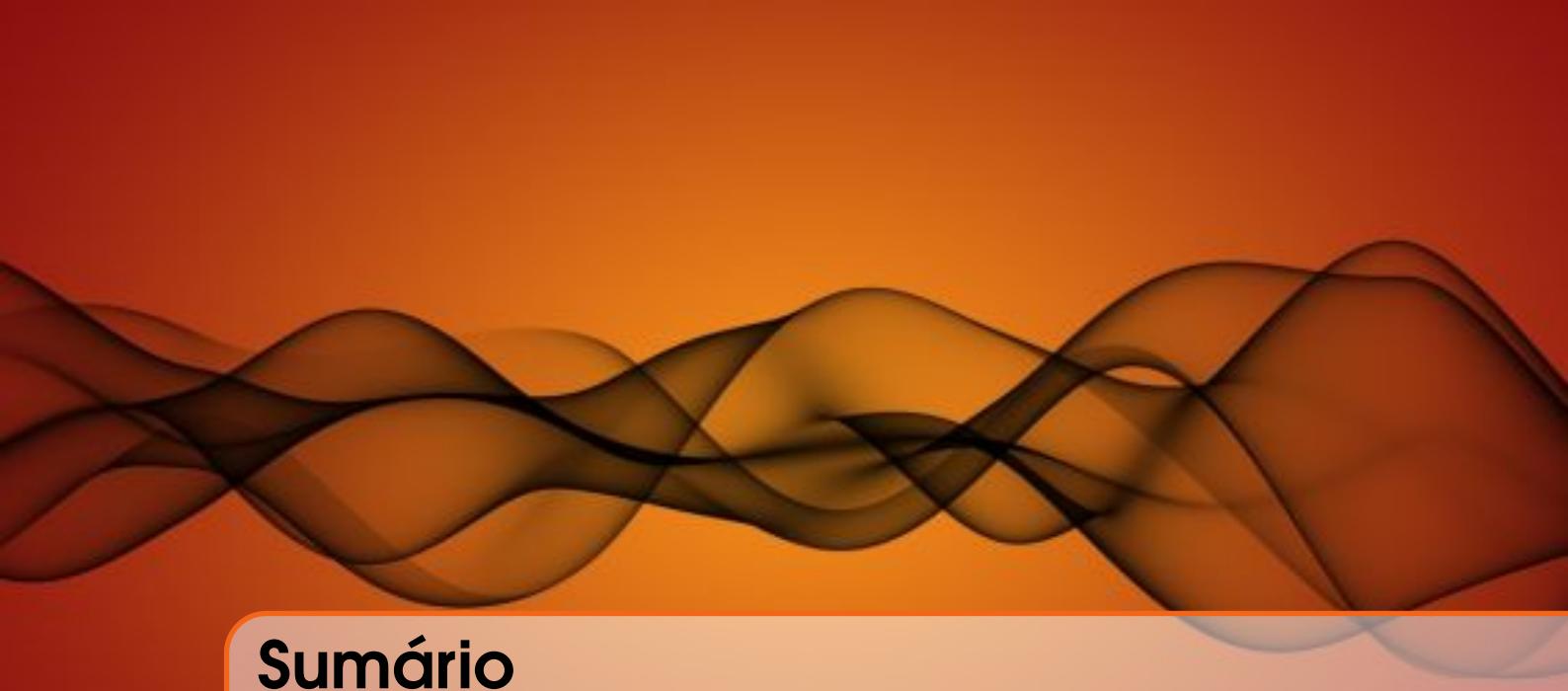
Observação: Requisitos mínimos significa a metade dos pontos

Copyright © 2023 Gabriel Costa Fassarella 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	Scala: A Programação Em Alta Performance	7
1.1	Introdução	7
1.2	História	8
1.3	JVM e SBT	9
1.4	Aplicações	10
1.4.1	Desenvolvimento Web	10
1.4.2	Big Data	11
1.4.3	Softwares Empresariais	12
2	Conceitos básicos da Linguagem Scala	15
2.1	Variáveis e constantes	15
2.1.1	var e val	15
2.2	Tipos de Dados Básicos	16
2.2.1	Char	16
2.2.2	String	17
2.2.3	Int	17
2.2.4	Float	17
2.2.5	Long	17
2.2.6	Short	18
2.2.7	Byte	18
2.2.8	Double	18
2.2.9	Boolean	18

2.3	Operadores e Expressões em Scala	18
2.3.1	Operadores Aritméticos	19
2.3.2	Operadores Lógicos	19
2.3.3	Operadores Comparadores	21
2.4	Entrada e Saída de Dados	22
2.4.1	Saída	22
2.4.2	Entrada	23
2.5	Estruturas Condicionais	23
2.5.1	if	23
2.5.2	match	24
2.6	Estruturas de Repetição	25
2.6.1	while	25
2.6.2	for	25
3	Programação em Scala	27
3.1	Funções	27
3.2	Classes	28
3.3	Listas	29
3.3.1	Operações	30
3.4	Tuplas	31
3.4.1	Operações	32
3.5	Set	32
3.6	Map	34
3.7	Array	35
4	Aplicações da Linguagem Scala	37
4.1	Operações Básicas	37
4.2	Calculadora	40
4.3	Regra do Trapézio	42
4.4	Bubble Sort	44
4.5	Quick Sort	46
4.6	Equação do Segundo Grau	50
4.7	Sistema Linear	52
5	Ferramentas existentes e utilizadas	55
5.1	Visual Studio Code (VS Code)	55
5.2	Scastie	58
6	Considerações Finais	61
	Bibliografia	65

1. Scala: A Programação Em Alta Performance

1.1 Introdução

Neste capítulo, trataremos sobre os principais aspectos da linguagem de programação Scala segundo os conceitos propostos pelos autores [OMS⁺21], [Sfr21] e [Wam21]. De acordo com [Sfr21] Scala é uma linguagem de programação funcional e orientada a objetos que durante os últimos anos vem crescendo em popularidade entre inúmeros desenvolvedores. O fato da Scala usar multiparadigmas faz com que sua eficiência, capacidade de lidar com projetos de grande porte e legibilidade sejam algumas de suas principais características. A linguagem usa dois tipos de paradigmas de programação: a programação orientada a objetos e ainda utiliza programação funcional.

Segundo [Sfr21]: "a abordagem orientada a objetos pode ser mais eficiente, mas propenso a erros. Ao usar o estado mutável, seu programa realocará sua memória: toda vez que ocorrer uma alteração, ele alterará os dados no lugar. No entanto, o estado de compartilhamento pode fazer com que seu aplicativo sofra problemas de inconsistência de dados devido a vários processos acessando e modificando a mesma porção de dados", ou seja, o uso da linguagem orientada a objetos pode apresentar uma boa performance, porém pode gerar uma boa quantidade de erros, já que é comum que nesse paradigma o programa receba diversas realocações de memória, mudando assim a localização dos dados, fazendo com que possam surgir problemas na data utilizada. Já a programação funcional pode garantir uma melhor legibilidade, já que por apresentar dados imutáveis o compartilhamento é mais seguro. Porém, esse paradigma apresenta uma menor performance, uma vez que o seu uso de memória costuma ser alto, já que a atualização de variáveis não é recorrente na programação funcional.

1.2 História

Scala foi fundada em 2002 pelo professor Martin Odersky (imagem abaixo) e sua equipe em uma escola política da Suíça, a École Polytechnique Fédérale de Lausanne (EPFL). Ainda em 1995, Odersky decidiu estudar o Java, e passou a se interessar em criar uma linguagem que usasse o paradigma de programação funcional. Isso fez com que ele se juntasse com Philip Wadler em uma equipe para criar uma linguagem funcional que fosse compilada para o Java Bytecode. Esse trabalho em conjunto, pouco tempo depois deu origem ao compilador Pizza.



Figura 1.1:

Fonte: [Twitter de Odersky](#)

O seu trabalho com o Pizza fez com que em 1997 ele entrasse em contato com a equipe de desenvolvedores da Sun Microsystems. O que permitiu a sua participação no desenvolvimento do Generic Java (GJ), um novo compilador javac. Segundo [VS09] em uma entrevista com o próprio Martin Odersky [disponível aqui](#): "Foi emocionante trabalhar em Java por causa de seu enorme impacto, mas também foi muito difícil porque Java é uma linguagem rica com muitos recursos que muitas vezes entram em conflito com extensões de maneiras imprevistas."

Em 1999, Odersky passou a lecionar na EPEL, e assim se sentiu livre para começar a criar o projeto de uma linguagem que unisse a programação orientada a objetos e programação funcional sem as limitações impostas pelo Java. Com essa ideia em mente, ele criou o Funnel, uma linguagem descrita por ele em [VS09] como: "lindamente simples, com muito poucos recursos de linguagem primitiva" criada para a utilização em redes funcionais. Porém, o uso da linguagem não se mostrou tão eficaz, já que o minimalismo apresentado pela linguagem trazia dificuldades para novos usuários, e para os mais experientes se tornava uma tarefa entediante ter que executar as mesmas linhas de código diversas vezes.

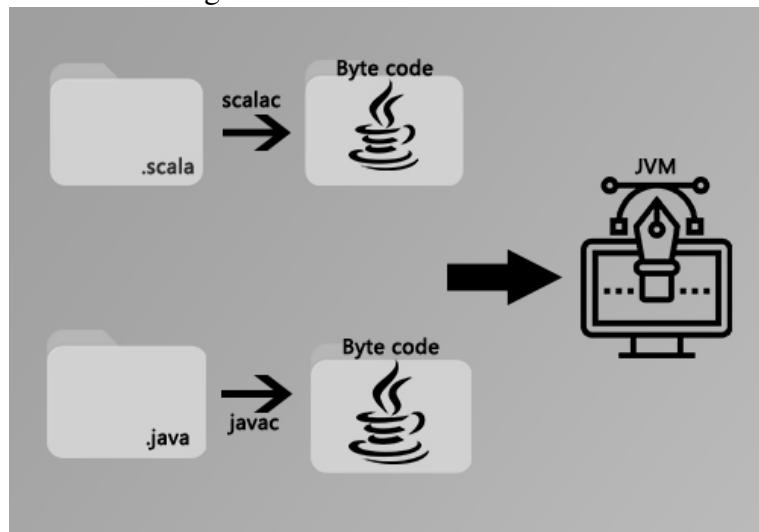
Tal cenário fez com que fosse necessária a criação de uma linguagem que apresentasse uma grande quantia de bibliotecas padrões. Isso fez com que Odersky começasse a

trabalhar na linguagem Scala em 2001, até o seu primeiro lançamento em 2003. Segundo Martin Odersky, Scala foi criado não com o intuito de ser uma extensão do Java, mas sim uma linguagem totalmente integrada com ele, já que Scala é traduzido para um Bytecode de java. Segundo Odersky em [VS09]: "Scala foi projetado para ser orientado a objetos e funcional. É uma linguagem puramente orientada a objetos no sentido de que todo valor é um objeto. Objetos são definidos por classes, que podem ser compostas usando composição mixin. Scala também é uma linguagem funcional no sentido de que toda função é um valor. As funções podem ser aninhadas e podem operar em dados usando correspondência de padrões."

1.3 JVM e SBT

Uma linguagem orientada a objetos com tipagem estática, como o Scala, tem como característica a verificação dos tipos de dados durante a compilação. Ou seja, se existe um dado do tipo float, apenas operações do tipo float serão permitidas com esse dado, evitando uma série de erros, uma vez que esses bugs podem ser contidos durante o processo de compilação. Porém para isso, é necessário utilizar a JVM (Java Virtual Machine), uma máquina virtual usada para executar uma sequência de instruções denominadas Bytecode.

Figura 1.2: Funcionamento JVM



Fonte: Autor

De acordo com [Sfr21]: "A JVM é uma máquina para realizar tarefas executando um conjunto bem definido de operações, chamadas de bytecode. Uma linguagem JVM visa traduzir seu código em executável Bytecode JVM, geralmente formado por vários arquivos com extensão *.class. Ao codificar em Java, você salva seus arquivos fonte com extensão *.java e usa o compilador javac para produzir um arquivo jar contendo o bytecode gerado", ou seja, quando um usuário está programando em Java, e deseja executar um arquivo, é necessário que o mesmo seja salvo utilizando a extensão .java, para que assim o compilador javac possa ser usado para criar um arquivo java que possua o bytecode que deverá ser traduzido para código de máquina nativo pela JVM, dessa forma executando as instruções desejadas. Algo semelhante ocorre com Scala, o usuário deve salvar o arquivo usando a

extensão .scala, isso faz com que o compilador scalac crie um arquivo scala que possua o bytecode, e por sua vez seja utilizado pela JVM para executar as instruções. Além desse processo, a JVM é responsável por outras funções, como verificar a integridade de um arquivo bytecode, organizar e a memória do código e gerenciar os processos.

A JVM foi criada com o intuito de ofertar ao usuário uma plataforma independente para a execução de seus programas, isso faz com que esses códigos possam ser executados em qualquer sistema operacional, desde que ele tenha suporte para a JVM. Esse cenário faz com que linguagens que utilizem a JVM, sejam extremamente populares na programação de multiplataformas.

Com isso, tornou-se necessária a criação de uma ferramenta que facilitasse a construção, empacotamento e compilação de códigos em Scala. Para isso foi criado o SBT (Simple Build Tool), de acordo com [Sfr21]: "sbt é a ferramenta de construção mais comum na comunidade. É um complexo, mas poderoso ferramenta que gerencia suas dependências e define o ciclo de construção do seu código", ou seja, ela é uma ferramenta associada a linguagem, responsável por gerenciar os projetos em Scala, podendo gerar desde documentações até executar alguns testes automatizados, além de outras operações necessárias para a construção de um código.

Utilizando o SBT é possível que o usuário possa automatizar tarefas mais comuns do Scala, fazendo com que o processo de desenvolvimento do código seja mais fácil e eficiente. Isso faz com que essa ferramenta seja extremamente utilizada pela comunidade Scala, e é considerada como uma ferramenta indispensável para os desenvolvedores que usam a linguagem.

1.4 Aplicações

Scala é uma línguagem extremamente moderna e versátil, que pode ser usada para inúmeras aplicações diferentes, seja pela sua segurança, performance, ampla variedade de bibliotecas ou compatibilidade com outras linguagens, fazendo com que diversas áreas diferentes possam optar pela utilização da Scala em seus códigos. Alguns exemplos de aplicações da linguagem são:

1.4.1 Desenvolvimento Web

Como demonstrado na figura abaixo, a área do desenvolvimento Web pode abranger uma vasta rede de categorias diferentes. Por esse motivo Scala costuma ser amplamente utilizada na produção de web aplicativos de alta performance. Isso ocorre pois Scala é uma linguagem de alto nível orientada a objetos, e seus projetos apresentam uma grande efetividade, uma vez que ela foi feita para ser executada em tempo real pelo compilador, acelerando assim o tempo de processamento do código.



Figura 1.3:
Fonte: [PNG Wing](#)

A compatibilidade do Scala com Java e outras linguagens de programação é um importante fator no uso da linguagem nessa área, uma vez que essa característica permite o uso de bibliotecas e frameworks java, podendo ainda aproveitar o uso da sintaxe breve apresentada pela linguagem. Tal situação possibilita uma maior liberdade e facilidade ao desenvolvedor na utilização do Scala em seus projetos.

Outro fator fundamental para a utilização do Scala em desenvolvimento web é a presença da modularidade e escalabilidade, que permitem a criação de projetos complexos. A programação assíncrona é extremamente importante para isso, uma vez que ela permite que os programadores criem códigos assíncronos de maneira segura.

1.4.2 Big Data

Como ilustrado na imagem abaixo, Big Data é uma área que envolve um imenso fluxo de dados diferentes. Por isso Scala é uma linguagem muito utilizada na manipulação de big data, podendo processar uma grande quantia de dados em tempo real. Isso ocorre porque a linguagem foi adaptada para lidar com um enorme volume de informações, por meio da programação funcional, modularidade e escalabilidade apresentada pela Scala. Isso ainda permite ao programador uma maior flexibilidade e uma programação mais abrangente, já que a Scala é uma linguagem de multiparadigmas.



Figura 1.4:
Fonte: [PNG Wing](#)

Além disso, a linguagem foi feita para ter um suporte de programação assíncrona e paralela, ou seja, o programa executa múltiplas tarefas de uma só vez, mas de maneiras diferentes. Isso permite que tarefas possam ser executadas em paralelo, permitindo o aumento da velocidade de processamento dos dados apresentados.

Scala ainda é uma linguagem de programação que apresenta uma sintaxe sólida e breve, isso permite que a aplicação de projetos para big data sejam extremamente simplificados, já que os programadores podem escrever menos código e realizar mais funções, acelerando o processo e custo de desenvolvimento, podendo ainda escrever códigos eficientes e de alto desempenho.

1.4.3 Softwares Empresariais

Como ilustrado na imagem abaixo, empresas costumam lidar com uma grande quantia de usuários gerenciando os seus sistemas de inúmeras maneiras. Por esse motivo o Scala pode ser usado para o desenvolvimento de softwares empresariais que exigem aplicativos escaláveis, uma vez que, em alguns casos, essas empresas necessitam de softwares que tratem com uma enorme quantia de dados e/ou tráfego de usuários, e isso permite que a linguagem acompanhe o crescimento da empresa, e como já dito antes, a imutabilidade garante a segurança dessa data.



Figura 1.5:
Fonte: [PNG Wing](#)

Como já citado anteriormente, a possibilidade da programação assíncrona e em paralelo permite que os dados sejam processados mais rapidamente. Tal cenário junto com a utilização da JVM (que permite uma otimização de desempenho) faz com que a linguagem tenha uma grande vantagem para o mercado, já que o fluxo de dados em empresas (especialmente nas de grande porte) é extremamente alto.

Além disso, o fato de que Scala é uma linguagem com multiparadigmas faz que ela seja uma extremamente versátil e adaptável, podendo ser usada para inúmeras necessidades que uma empresa exige diariamente, permitindo ainda a construção de códigos eficientes e alto desempenho.

2. Conceitos básicos da Linguagem Scala

Os livros básicos usados para o estudo da Scala são [OMS⁺²¹], [Sfr21] e [Wam21]. Neste capítulo serão apresentados alguns conceitos básicos necessários para a programação em Scala, como o uso de variáveis, operadores e funções básicas.

2.1 Variáveis e constantes

Segundo [OMS⁺²¹] variável é um dos principais conceitos da programação, em essência uma variável é um espaço de memória responsável por armazenar um valor, podendo ser de diferentes tipos, como números, texto, valores booleanos e tipos de dados mais complexos. Para declarar uma variável no código é necessário definir se é uma val (constante) ou uma var, seu nome, o tipo de dado que ela irá armazenar e por último o seu valor. Uma variável é definida da seguinte forma:

```
var Nome : Tipo = valor
```

é muito importante ressaltar que a declaração do tipo da variável em Scala é opcional, porém é extremamente recomendada. Isso ocorre porque além de Scala ser uma linguagem de programação extremamente tipada (toda a variável deve ter seu tipo definido), ela possui inferência de tipo, ou seja, o compilador pode determinar automaticamente o tipo da variável se baseando no dado atribuído a ela.

2.1.1 var e val

[Sfr21] reitera que em Scala, uma var é um tipo de variável mutável, ou seja, ela pode ter o seu valor alterado a qualquer momento.

Exemplo de uma aplicação de var:

```
scala> var msg: String = "Olá, bom dia!"
```

```
var msg: String = Ola, bom dia!

scala> msg = "Ola, boa tarde!"
msg: String = "Ola, boa tarde"
```

Uma val funciona de maneira contrária a uma var, já que a val representa uma constante, uma vez que ela possui valor imutável, ou seja o valor de uma val não pode ser alterado.

Exemplo de val:

```
scala> val msg: String = "Ola, bom dia!"
val msg: String = Ola, bom dia!
```

Vale citar que caso o usuário tente alterar o valor de uma val (nesse caso msg), um erro será mostrado.

Exemplo:

```
scala> msg = "Ola, boa tarde!"
-- [E052] Type Error: -----
1 |msg = "Ola, boa tarde!"  
|-----  
|  
| Reassignment to val msg  
|  
| longer explanation available when  
compiling with '-explain'  
1 error found
```

2.2 Tipos de Dados Básicos

Nessa seção do capítulo abordaremos os inúmeros tipos principais de dados básicos existentes em Scala, ou seja o tipo do valor que será armazenado em um espaço de memória (variável).

2.2.1 Char

Na linguagem Scala, o Char é um tipo primitivo de dado numérico usado para a representação de um único caractere qualquer em UNICODE. O Char é definido se utilizando de duas aspas simples ("") com apenas um caractere em seu interior. Exemplo:

```
scala> var car: Char = 'x'
var car: Char = x

scala> car = 'X'
car: Char = X
```

No exemplo acima é declarada uma variável chamada car do tipo Char, e nela é atribuído o valor 'x'. Ainda é válido observar que o valor de car é alterado para 'X', ou seja um Char maiúsculo é diferente de um Char minúsculo.

2.2.2 String

Em Scala, a String é um tipo de dado usado para representar um sequência de caracteres. Na linguagem, a string é um tipo de dado imutável, ou seja, ela não pode ser alterada. Isso faz com que operações com esse tipo de dado envolvam a criação de uma nova string.

Exemplo:

```
scala> var ola = "Ola, mundo!"
var ola: String = Ola, mundo!
```

Além disso, caso o usuário deseje utilizar aspas no interior da String, no momento da declaração é necessário usar aspas triplas ()

```
scala> var msg = """A mensagem dizia "Ola
, mundo!""""
var msg: String = A mensagem dizia "Ola,
mundo!"
```

2.2.3 Int

Int em Scala é um tipo de dado primitivo que se refere aos números inteiros positivos e negativos pertencentes a um intervalo suportado pela memória que vai até 32 bits. Com esse tipo de dado é possível realizar inúmeras operações aritméticas.

```
scala> var x: Int = 5
var x: Int = 5

scala> var y: Int = 3
var y: Int = 3

scala> var res: Int = x + y
var res: Int = 8
```

No código acima, foi declarado duas variáveis (x e y) do tipo Int, e é guardada a soma das duas variáveis em outra variável também do tipo Int de nome res. Vale lembrar que o funcionamento das operações serão tratadas no tópico 2.3 deste capítulo.

2.2.4 Float

Na linguagem Scala, o Float é um tipo de dado primitivo que representa um valor real de ponto flutuante, com uma precisão de cerca de 7 casas dígitos significativos que vai até valores de 32 bits. Para que o compilador não confunda um valor do tipo float com o tipo double, é necessário colocar o sufixo 'f'. Isso não é obrigatório, porém auxilia o compilador a identificar o tipo do dado.

```
scala> var pi: Float = 3.141593f
var pi: Float = 3.141593
```

2.2.5 Long

Em Scala, o Long representa assim como o Int um tipo inteiro porém longo, ou seja ele atinge valores que vão até os 64 bits. Assim como o tipo Float, também é necessário usar

um sufixo para que o compilador identifique o tipo do dado iniciado, neste caso o sufixo que deverá ser utilizado é o 'l'.

```
scala> var num: Long = 345769341
      var num: Long = 34576934
```

2.2.6 Short

Semelhante ao Long, o Short também representa números inteiros porém mais curtos, armazenando até 16 bits.

```
scala> val numS: Short = 6436
      val numS: Short = 6436
```

O Short é extremamente útil para operações com valores inteiros inferiores ao intervalo suportado pelo Int, além disso o Short ocupa menos espaço de memória que o Int.

2.2.7 Byte

Como o próprio nome, o Byte é capaz de armazenar dados de tamanho ainda menor que o Short, chegando a uma quantia equivalente a 8 bits, uma vez que 1 byte equivalem a 8 bits.

```
scala> val dia: Byte = 21
      val dia: Byte = 21
```

Assim como o Short, o Byte é utilizado em operações de números pequenos, porém é importante lembrar que os valores do Byte pertencem a um intervalo extremamente pequeno (-128 a 127).

2.2.8 Double

Em Scala o Double é utilizado para representar valores reais de ponto flutuantes que vão até 64 bits, isso faz com que a precisão dos valores seja aumentada.

```
scala> val e: Double = 2.71828182845995
      val e: Double = 2.71828182845995
```

2.2.9 Boolean

Na linguagem Scala o Boolean, representa os valores booleanos, ou seja representam valores lógicos de verdadeiro (true) ou falso (false). Vale lembrar que o funcionamento das operações lógicas serão melhor abordadas na próxima seção desse capítulo.

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

2.3 Operadores e Expressões em Scala

Nessa seção abordaremos o funcionamento das operações aritméticas, comparativas e lógicas na linguagem Scala, assim como exemplos demonstrando as suas utilizações.

2.3.1 Operadores Aritméticos

Em Scala, os operadores aritméticos são os mesmos que outras linguagens de programação geralmente possuem.

são elas:

- Soma (+): utilizada para executar a soma entre valores.
- subtração (-): usada para efetuar a subtração entre valores.
- multiplicação (*): usada para realizar a multiplicação entre valores.
- Divisão (/): utilizada na execução da divisão entre valores.
- Módulo (%): usada para obter o resto de divisão entre valores.

Com esses operadores aritméticos é possível realizar inúmeras operações matemáticas entre valores quaisquer, inclusive entre variáveis que guardem números.

Exemplo:

```
scala> var x: Int = 5
var x: Int = 5

scala> var y: Int = 3
var y: Int = 3

scala> x + y
val res0: Int = 8

scala> x - y
val res1: Int = 2

scala> x * y
val res2: Int = 15

scala> x / y
val res4: Int = 1

scala> x % y
val res5: Int = 2
```

é importante observar que os resultados das operações são guardados em variáveis do tipo Int, fazendo com que valores de tipos distintos não sejam aceitos. é possível observar isso na operação de divisão entre x e y, uma vez que o valor da divisão de 5 por 3 é um dízima periódica de valor aproximadamente 1.67, ou seja, um ponto flutuante. Porém em Scala, quando se deseja guardar um valor flutuante em uma variável do tipo inteiro, é armazenada apenas a parte inteira do número, neste caso o valor 1.

2.3.2 Operadores Lógicos

De acordo com [Wam21] os operadores lógicos são símbolos usados para realizar operações lógicas entre valores booleanos, devolvendo assim um novo valor booleano.

são eles:

- Operador AND (&&): realiza a operação "e"entre valores.
- Operador OR (||): executa a operação "ou"entre valores.

- Operador NOT (!): realiza a negação de um valor ou operação.

Para utilizar esses operadores é necessário primeiramente entender um pouco do funcionamento da lógica booleana.

Operador AND

Considerando duas premissas, P e Q que podem assumir tanto verdadeiro quanto falso, temos que a tabela verdade do operador and funciona da seguinte forma:

P	Q	$P \wedge Q$
Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Falso
Falso	Verdadeiro	Falso
Falso	Falso	Falso

Tabela 2.1: Tabela verdade do operador "and".

é importante lembrar que quando são feitas operações usando o operador "and" entre duas premissas, só é verdadeiro quando ambas as premissas são verdadeiras.

Operador OR

Considerando duas premissas, P e Q que podem assumir tanto verdadeiro quanto falso, temos que a tabela verdade do operador or funciona da seguinte forma:

P	Q	$P \vee Q$
Verdadeiro	Verdadeiro	Verdadeiro
Verdadeiro	Falso	Verdadeiro
Falso	Verdadeiro	Verdadeiro
Falso	Falso	Falso

Tabela 2.2: Tabela verdade do operador "or".

Vale ressaltar que quando se utiliza o operador "or", o retorno é falso apenas quando as premissas P e Q também são falsas.

NOT

Considerando uma premissa, P que pode assumir tanto verdadeiro quanto falso, temos que a tabela verdade do operador not funciona da seguinte forma:

P	$\neg P$
Verdadeiro	Falso
Falso	Verdadeiro

é possível perceber que quando se utiliza o operador "not" o valor da premissa P é invertido.

Exemplo de algumas operações lógicas em Scala:

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

```

scala> val b: Boolean = false
val b: Boolean = false

scala> val c: Boolean = true
val c: Boolean = true

scala> val d: Boolean = false
val d: Boolean = false

scala> a && b
val res0: Boolean = false

scala> a && c
val res1: Boolean = true

scala> a || b
val res2: Boolean = true

scala> b || d
val res3: Boolean = false

scala> !a
val res4: Boolean = false

```

2.3.3 Operadores Comparadores

Conforme afirmado por [OMS⁺²¹], os operadores de comparação em Scala são usados para comparar valores, retornando um valor booleano verdadeiro ou falso.

- Maior que (>): Retorna verdadeiro se o número da esquerda for maior que o da direita.
- Menor que (<): Retorna verdadeiro se o número da esquerda for menor que o da direita
- Maior ou igual a (>=): Retorna verdadeiro se o número da esquerda for maior ou igual ao da direita.
- Menor ou igual a (<=): Retorna verdadeiro se o número da esquerda for menor ou igual ao da direita.
- Igual a (==): Retorna verdadeiro se o número da esquerda for igual ao da direita.
- Diferente de (!=): Retorna verdadeiro se o número da esquerda for diferente do valor da direita.

Exemplos de operações com operadores de comparação:

```

scala> val a: Int = 9
val a: Int = 9

scala> val b: Int = 3
val b: Int = 3

```

```

scala> val c: Int = 9
val c: Int = 9

scala> a > b
val res0: Boolean = true

scala> a < b
val res1: Boolean = false

scala> b >= c
val res2: Boolean = false

scala> a >= c
val res3: Boolean = true

scala> a == c
val res4: Boolean = true

scala> b != c
val res5: Boolean = true

```

2.4 Entrada e Saída de Dados

Em programação, a entrada e saída de dados se refere a como um código é capaz de receber dados de um usuário e como esse mesmo programa é capaz de devolver alguma informação.

Para demonstrar esses conceitos será necessário o uso de uma IDE para que seja possível criar e executar um código de maneira efetiva.

2.4.1 Saída

Conforme dito por [OMS⁺²¹], em Scala existem duas formas para mostrar a saída de um dado para o usuário, muito semelhantes as já existentes em outras linguagens. São elas os comandos `print` e `println`, sendo que o primeiro mostra uma sequência de caracteres escrito em seu interior porém não quebra a linha, já o segundo também mostra a sequência de caracteres porém quebrando a linha.

```

println("Haverá uma quebra de linha no
       final desse texto!")
print("É possível escrever uma frase ")
print("utilizando dois prints diferentes
      \n")

```

Note que ainda é possível quebrar a linha utilizando o comando `\n` dentro do próprio `print`.

2.4.2 Entrada

Em Scala, para receber dados de um usuário é necessário importar uma biblioteca chamada 'scala.io.StdIn', ela permite que o usuário possa dar a entrada de dados em um código.

```
import scala.io.StdIn.readLine

object hello {
    def main(args: Array[String]): Unit = {
        println("Para qual time
voce torce? ")
        val time : String =
            readLine ()
    }
}
```

2.5 Estruturas Condicionais

Segundo [Wam21], em programação no geral, as estruturas condicionais são construções que permitem o programa tomar decisões se baseando em condições estabelecidas pelo programador. Essas condições podem retornar valores booleanos, ou uma variável ou número que pode ser avaliado e retornar um valor booleano. A estrutura condicional tem o papel de avaliar essas condições, e dependendo do retorno dado, executar ou não um certo trecho do código. Em Scala existem duas principais estruturas condicionais, são elas: o 'if' e o 'match'.

2.5.1 if

A estrutura condicional if é semelhante a de outras linguagens de programação, de acordo com [Sfr21], nela é apresentada uma condição que se retornar um valor verdadeiro, o código apresentado em seu corpo é executado.

```
if(condicao) {
    //codigo que deseja ser executado se
    //condicao for true
}
```

A estrutura if ainda pode ter uma variação também presente em outras linguagens de programação: o if - else. Essa variação permite que o código possa tomar uma ação alternativa se a condição imposta no if retornar falso.

```
if(condicao) {
    //codigo que deseja ser executado se
    //condicao for true
}

else {
    //codigo que deseja ser executado se
    //condicao for false
}
```

```
}
```

Exemplo de uma aplicação do if - else em um código:

```
val num = 4

if (num % 2 == 0) {
    println("O numero e par")
}

else {
    println("O numero e impar")
}
```

[Running] O numero e par

O código apresentado é responsável por verificar se um número qualquer é par ou ímpar, é importante notar que a condição 'num % 2 == 0' retornou verdadeiro e seu código foi executado, caso a condição retornasse falso, o código presente no corpo de else seria executada. é importante ressaltar ainda que o comando 'println()' é utilizado para mostrar na tela do usuário a sequência de caracteres escritas em seu interior.

2.5.2 match

Como dito por [OMS⁺²¹], a estrutura match em Scala é usada para testar uma sequência de padrões e/ou condições, e de acordo com o resultado, realizar uma ação correspondente. A estrutura do match funciona da seguinte forma:

```
value match {
    case padrao1 => fazer1
    case padrao2 => fazer2
    case padrao3 => fazer3
    case _ => AcaoPadrao
}
```

No exemplo acima, 'value' é o valor que está sendo analisado, 'padrao1', 'padrao2' e 'padrao3' são condições que será verificadas, e 'fazer1', 'fazer2' e 'fazer3' são as ações que serão tomadas caso a condição seja verificada, e por fim o 'case_' corresponde a ação tomada caso nenhuma das anteriores sejam verificadas.

```
val num = "Pedro"

num match {
    case "Pedro" => println("O nome e Pedro")
    case "Kevin" => println("O nome e Kevin")
    case "Enzo" => println("O nome e Enzo")
    case _ => println("E outro nome")
}
```

```
[Running] 0 nome e Pedro
```

Observe que o valor 'num' é testado no interior no match e é verificado no primeiro caso a condição, com isso é executada a ação desejada, nesse caso mostrar para o usuário o nome do individuo.

2.6 Estruturas de Repetição

Como citado por [Wam21], em programação, uma estrutura de repetição é uma construção que permite o programa executar um trecho do código diversas vezes enquanto uma determinada condição se provar verdadeira. Em Scala existem duas estruturas de repetição principais, o for e o while.

2.6.1 while

A estrutura while em Scala funciona de forma semelhante ao while presente em outras linguagens de programação, segundo [Wam21], é dada uma condição e enquanto essa condição for verificada, o código presente em seu corpo será executado. A estrutura do while é dada da seguinte forma:

```
while (condicao) {
    //corpo da estrutura com o codigo
}
```

Exemplo de um contador utilizando a estrutura de repetição while:

```
var i = 0

while (i <= 5) {
    println(i)
    i += 1
}
[Running] 0
1
2
3
4
5
```

O código acima apresenta uma variável índice i que é iniciada com 0, logo a condição 'i <= 5' é verificada e o loop é iniciado enquanto a condição for verdadeira, sempre acrescentando 1 na variável contadora i.

2.6.2 for

A estrutura de repetição for é usada para interagir com uma sequência de valores, ou realizar uma série de repetições uma quantia específica de vezes. A estrutura dela é dada da seguinte forma:

```
for(indice <- inicio to fim by passo) {
    //corpo com o codigo
}
```

A estrutura for deve apresentar uma variável índice (que funcionará como um contador) que deve apresentar um início e um fim, sendo que cada repetição será acrescentado um valor ao índice baseado no passo imposto pelo programador, até que esse índice atinja o seu fim. Vale lembrar que caso não seja imposto nenhum passo, por padrão ele terá o valor 1.

Exemplo de um contador usando o for:

```
for(i <- 0 to 10 by 2) {  
    println("Mensagem " + i)  
}  
  
[Running] Mensagem 0  
Mensagem 2  
Mensagem 4  
Mensagem 6  
Mensagem 8  
Mensagem 10
```

No código acima o for é iniciado com um índice i começando 0 indo até o 10 somando de 2 em 2, e a cada repetição é executado o código presente em seu corpo, que no caso é mostrar uma mensagem para o usuário.

3. Programação em Scala

Neste capítulo avançaremos mais ainda na programação em Scala, abordando temas relacionados as classes, funções, listas, vetores e entre outras estruturas apresentadas na linguagem, seguindo os autores [OMS⁺21], [Sfr21] e [Wam21].

3.1 Funções

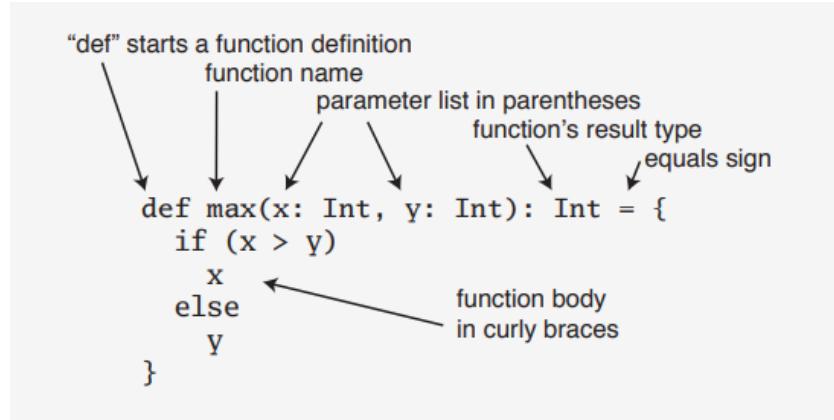
Em programação, uma função é um bloco de código responsável por executar uma instrução específica, recebendo um ou mais valores (parâmetros) como entrada podendo retornar ou não um valor específico.

Funções costumam ser usadas para reduzirem o tamanho do código ou torna-lo mais organizado, gerenciáveis e reutilizáveis. Criando essas funções, é possível encapsular um trecho específico do código, permitindo o programador realizar chamadas da função em outras partes do programa. Isso garante que o código seja mais fácil de ler, de testar e também de manter.

Em Scala, funções são estruturas de primeira classe, ou seja, podem ser atribuídas a variáveis, passadas para outras funções como um argumento e pode ainda ser retornada como valor de outra função.

Na figura abaixo retirada o livro "programming in scala" de [OMS⁺21] é possível identificar a estrutura de uma função em Scala. Segundo [OMS⁺21] as funções são sempre iniciadas usando a palavra-chave "def", seguido pelo seu nome e por um parênteses com seus parâmetros e tipagem. Após isso é necessário especificar o tipo de dado retornado pela função, em seguida basta escrever o corpo que vai apresentar o código dessa função.

Figura 3.1: Estrutura Função

Fonte: [OMS⁺²¹]

No exemplo acima existe uma função chamada max que retorna um valor int, ainda possui dois parâmetros: x e y, ambos com tipagem int. Nessa função, o retorno será decidido por alguma operação dentro das chaves, nesse caso um if, que escolhe entre x e y.

Uma vez criada a função basta chama-la no código principal, para isso é necessário escrever o seu nome e os parâmetros desejados:

Neste caso:

```
max(5, 7)
```

```
[Running] 7
```

Vale lembrar que todo o código Scala é rodado dentro de uma função denominada main, o mesmo ocorre em outras linguagens como C por exemplo. A estrutura do main é dada da seguinte forma:

```

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

Para realizar um hello world em Scala é necessário primeiro declarar uma classe, para isso se utiliza a expressão "object [nome da classe]", dessa forma podemos criar a função denominada main, onde será rodado o código em Scala. Usaremos como parâmetro o args que é um array de strings, dessa forma será passado uma série de argumentos de linhas de comando para o código. Além disso é utilizado o tipo de dado Unit como saída, indicando uma saída vazia. Após isso basta utilizar o println como já visto antes.

3.2 Classes

Segundo [Sfr21] classes são estruturas fundamentais para linguagens orientadas a objetos, uma vez que elas são utilizadas para definir um objeto. Elas são extremamente usadas para encapsular dados e alguns comportamentos relacionados entre si.

Para definir uma classe basta usar a palavra-chave 'class' seguida de seu nome. Exemplo:

```
class individuo(nome: String, idade: Int) {
    def saudacao(): Unit = {
        println(s"Ola, meu nome e $nome e
                eu tenho $idade anos.")
    }

    def despedida(nome2: String): Unit = {
        println(s"Adeus $nome2, foi um
                prazer te conhecer!")
    }
}

def main(args: Array[String]) = {
    val julio = new individuo("Julio" , 15)

    julio.saudacao()
    julio.despedida("Gabriel")
}

[Running]
Ola, meu nome e Julio e eu tenho 15 anos.
Adeus Gabriel, foi um prazer te conhecer!
```

Note que no exemplo dado inicialmente foi criada a classe "individuo" junto de suas entradas, nesse caso o nome e a idade do individuo. Após isso, foram criadas algumas funções, nesse caso chamadas de métodos, sendo esses o método saudacao, que irá realizar um print com o nome e idade, e o outro método sendo o despedida, que irá receber uma entrada, sendo essa um novo nome que será printada no código. Já no main é realizado a declaração da classe e suas respectivas chamadas.

3.3 Listas

Segundo [Sfr21] em Scala, listas são uma reunião de elementos/dados que são imutáveis de um mesmo tipo. Ou seja, uma vez criada uma lista, não é possível alterá-la, porém todos os métodos possíveis de adicionar ou remover elementos de uma lista retornam uma nova lista com as alterações efetuadas.

Para criar uma lista, é necessário utilizar a classe list ou uma sintaxe construtora de listas.

Exemplo de uma lista com 3 elementos:

```
def main(args: Array[String]) = {
    val lista1 = List(1, 2, 3)

    val lista2 = 4 :: 5 :: 6 :: Nil
```

```

    val lista3 : List [Int] = List (7, 8, 9)

    println(lista1)

    println(lista2)

    println(lista3)
}

[Running]
List(1, 2, 3)
List(4, 5, 6)
List(7, 8, 9)

```

Note que na primeira linha foi necessário utilizar a classe "List" com o objetivo de criar a lista com os 3 números inteiros solicitados. Já na segunda foi utilizada uma sintaxe construtora de lista, sendo que nela o operador "::" adiciona elementos a lista e o operador "Nil" é usado para representar uma lista vazia. E por último na terceira linha também é usada a classe "List", porém nesse caso é identificado o tipo dos valores da lista.

Ainda é possível mostrar como saída os valores da lista por índice, para isso basta identificar a posição que deseja mostrar para o usuário. Vale lembrar que a contagem dos índices sempre começa pelo valor 0.

Exemplo:

```

def main(args: Array[String]) = {
    val lista1 = List(1, 2, 3)

    println(lista1(0))
    println(lista1(1))
    println(lista1(2))
}

[Running]
1
2
3

```

3.3.1 Operações

Em Scala é muito comum o uso das operações com listas, visto a sua alta capacidade de utilização, permitindo o programador efetuar uma série de manipulações diferentes de maneira simples e extremamente eficiente.

Considerando as listas: lista1 = (1,2,3) e a lista2 = (4, 5, 6), veja alguns exemplos de operações com essas listas:

```

// concatenacao
val lista3 = lista1 ++ lista2
[Running] List(1, 2, 3, 4, 5, 6)

```

```
// adicionando elementos
val lista4 = lista1 :+ 4
[Running] List(1, 2, 3, 4)

// removendo elementos
val lista5 = lista1.filterNot(_ == 3) // remove o elemnto 3
[Running] List(1, 2)

// mapeamento
val lista6 = lista1.map(_ * 2) // multiplica a lista por 2
[Running] List(2, 4, 6)

// reduzindo elementos
val red = lista2.reduce(_ + _) // soma todos os elementos
[Running] 15
```

3.4 Tuplas

Segundo [OMS⁺²¹]: "Assim como as listas, as tuplas são imutáveis, mas ao contrário das listas, as tuplas podem conter diferentes tipos de elementos". Ou seja, não é possível alterar os valores de uma tupla, porém diferente das listas, uma mesma tupla pode conter um dado do tipo inteiro e outro dado do tipo string por exemplo.

Para criar uma tupla em Scala é necessário iniciá-la usando parênteses separando os seus elementos por vírgula.

Exemplo de uma tupla composta por uma string e um float:

```
def main(args: Array[String]) = {
    val tupla1 = ("Ola!", 3.14)

    println(tupla1)
}

[Running] (Ola!, 3.14)
```

Os elementos do interior de uma tupla podem ser acessados por meio da notação "NomeDaTupla._Posição", sendo que a posição diferente de uma lista começa pelo número 1.

```
def main(args: Array[String]) = {
    val tupla1 = ("Ola!", 3.14)

    println(tupla1._1)
    println(tupla1._2)
}
```

```
[Running]
Ola!
3.14
```

3.4.1 Operações

Assim como as listas, as tuplas apresentam algumas operações possíveis que podem ser extremamente úteis durante a programação. Considerando a tupla val tupla1 = ("Ola!", 3.14), veja alguns exemplos das operações mais comuns envolvendo tuplas em Scala.

```
// Desestruturar tupla
val (msg, pi) = minhaTupla // atribui o
                            valor da tupla a variáveis individuais

println(msg)
println(pi)

[Running]
Ola!
3.14

// Concatenar tuplas
val tupla2 = ("x", 2)
val tupla3 = tupla1 ++ tupla2
println(tupla3)

[Running] (Ola!,3.14,x,2)

// Verificar Tamanho
val range = tupla1.size
println(range)

[Running] 2

// Transformar tupla em lista
val tupla1 = ("Ola!", "mundo")
val lista = tupla1.toList
println(lista)

[Running] List(Ola!, mundo)
```

3.5 Set

De acordo com [OMS⁺²¹], em Scala um set é uma estrutura responsável por agrupar um conjunto de elementos porém sem permitir uma "duplicata" dos mesmos, ou seja, cada elemento presente em um set é único e não possui uma cópia de si mesmo. O conteúdo de

um set é armazenado em uma ordem não definida e não possui nenhum índice associado a ele.

Para se implementar um set é necessário usar uma estrutura de dados chamada de "hash table". Isso permite que muitas operações como as de adição, remoção e verificação possam ser executadas constantemente, fazendo com que os sets sejam extremamente eficazes na manipulação de big data.

Exemplo:

```
val materias = Set("fisica", "calculo", "programacao")

println(materias)

[Running] Set(fisica, calculo,
programacao)
```

Em Scala, a estrutura set apresenta 2 tipos, o tipo mutável e o tipo imutável. O primeiro como o próprio nome diz não podem ser mudados depois de criados, ou seja, qualquer operação de adição ou remoção de elementos, por exemplo, feita em um set imutável retorna um novo set para o usuário. Já o set mutável é o oposto, ou seja, são estruturas que podem ser modificadas depois de sua criação, porém para cria-las é necessário utilizar a biblioteca "scala.collection.mutable.Set".

Operações com set imutável:

```
// criando set
val seq = Set(1, 2, 3)

// adicionando elemento
val newseq = seq + 4

// removendo elemento
val remseq = seq - 3

// unindo set
val newset = Set(4, 5, 6)
val uniset = seq ++ newset

// verifica a existencia de um elemento
val existe1 = seq.contains(1)

// verifica o tamanho do set
val range = seq.size
```

Operações com set mutável:

```
// importando biblioteca
import scala.collection.mutable.Set

// criando set vazio
val seq = Set[Int]()
```

```

// adicionando elemento
seq += 7
seq += 8
seq += 9

// removendo elemento
seq -= 9

// atualizando elemento
seq.update(7, 3) // substitui o 7 pelo 3

// verificando a existencia de um elemento
seq.contains(8)

// limpando set
seq.clear()

```

3.6 Map

Em Scala, "map" é uma coleção de elementos que armazena chaves e valores relacionados. É um conjunto de dados extremamente eficiente para realizar consultas, pesquisas e operações se baseando nas chaves.

Como criar um map em Scala:

```

val map = Map("key1" -> 1, "key2" -> 2, "key3" -> 3)

println(map("key1"))

[Running] 1

```

Para criar uma estrutura map em Scala, basta utilizar a classe "Map", e indicar as chaves e longo em seguida o valor específico dessa chave.

Algumas operações com map:

```

val map = Map("key1" -> 1, "key2" -> 2, "key3" -> 3)

// update: retorna um novo map com
// valores atualizados
map + (key4 -> 4)

// remove: retorna um novo map sem o par
// da chave
map - "key1"

// contains: verifica se o map possui uma
// chave

```

```

map.contains(key2)

//isEmpty: verifica se o map esta vazio
map.isEmpty

//size: retorna o tamanho do map
map.size

//keySet: retorna todas as chaves do map
map.keySet

//values: retorna todos os valores do map
map.values

```

3.7 Array

Segundo [OMS⁺²¹], em Scala um array é um conjunto de elementos armazenados e uma sequência fixa, e diferente das outras coleções imutáveis como set e lista, é possível alterar os valores presentes em um array mesmo após a sua criação, ou seja, o array é uma estrutura mutável.

Em Scala, todo array apresenta uma estrutura um tamanho fixo determinada determinado na sua criação, sendo que todos os seus elementos devem apresentar o mesmo tipo de dado, ou seja, array é uma estrutura de tipo homogêneo.

Como criar um array:

```

val array1: Array[Int] = Array(1,2,3)

println(array1(2))

[Running] 3

```

Para criar um array basta utilizar a classe e indicar os seus elementos.

Algumas operações:

```

//criando array
val array1: Array[Int] = Array(1,2,3)

//acessa o valor do indice informado
array1(0)

//modifica elemento no indice informado
array1 = 6

//tamanho do array
array1.length

```


4. Aplicações da Linguagem Scala

Neste capítulo será abordado algumas simples aplicações da linguagem Scala, apresentando o código fonte junto de imagens com sua aplicação e resultados assim como uma breve descrição do código.

4.1 Operações Básicas

O código que será abordado nessa seção se trata de uma simples implementação de uma calculadora do volume de um cilindro em Scala. Esse algoritmo exige ao usuário a entrada dos dados necessários para o cálculo (raio e altura) por meio de um menu interativo, e retorna ao mesmo o valor da área do cilindro.

Código fonte:

```
import scala.io.StdIn

object CalculadoraCilindro {
    def main(args: Array[String]): Unit = { // 
        Definindo o main, local em que o codigo sera
        rodado
        var continuar = true // Condicao de
        existencia do while

        while (continuar) {
            println("1. Calcular volume do
                    cilindro") // Menu
            println("2. Sair")
            val op = StdIn.readInt() // Dando
            entrada na opcao
```

```

        op match {
            case 1 => // Entrada dos
                        dados do cilindro
                println("Digite o raio do
                        cilindro:")
                val raio = StdIn.
                    readDouble()
                println("Digite a altura
                        do cilindro:")
                val altura = StdIn.
                    readDouble()
                val volume = calcVol(raio
                    , altura)
                println(s"O volume do
                        cilindro e: $volume")

            case 2 => // Fim do
                        codigo
                continuar = false
                println("Encerrando o
                        programa.")

            case _ => // Erro na
                        entrada
                println("Opção invalida.
                        Por favor, tente
                        novamente.")
        }
        println()
    }

    def calcVol(raio: Double, altura: Double): Double
    = { // Calculo do volume
        val areaB = 3.14 * raio * raio
        val volume = areaB * altura
        volume // Retorna o volume
    }
}

```

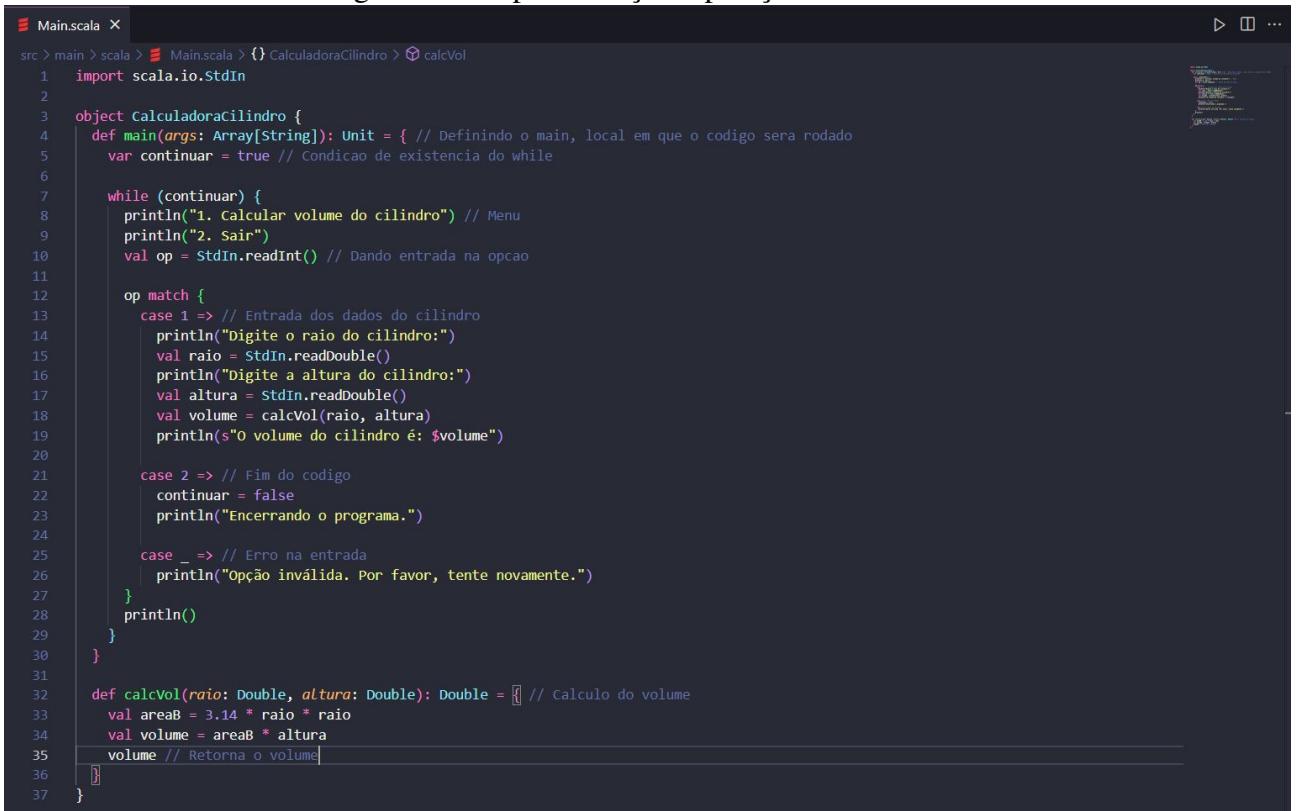
- Inicialmente é importada a biblioteca responsável pela entrada de dados.
- Em seguida é criado o main, área principal no qual o código é rodado.
- Após isso é mostrado o menu ao usuário, exigindo ao mesmo uma entrada. Vale lembrar que esse menu é criado dentro de um while, para caso o usuário deseje calcular a área de mais de um cilindro.
- Após isso é criada uma estrutura match para os casos apresentados na leitura do

menu. A primeira opção ocorre caso op seja 1, com isso são dadas as entradas do cilindro. A segunda opção é para caso op seja 2, quebrando assim o while e finalizando o código. E por último, é casa op seja qualquer outro valor, mostrando assim o menu novamente para o usuário.

- Com isso, caso op seja 1, será chamada a função de calculo de volume, que recebe o raio e a altura, podendo assim calcular e retornar o valor do volume, e em seguida o mostrando ao usuário.

Para a implementação desse algoritmo foi utilizada a Ide Vscode, para isso foi criado um projeto e escrito o código. Veja as imagens da implementação do programa e dos resultados obtidos ao compilar:

Figura 4.1: Implementação Operações



```

Main.scala x
src > main > scala > Main.scala > () CalculadoraCilindro > calcVol
1 import scala.io.StdIn
2
3 object CalculadoraCilindro {
4   def main(args: Array[String]): Unit = { // Definindo o main, local em que o código sera rodado
5     var continuar = true // Condicao de existencia do while
6
7     while (continuar) {
8       println("1. Calcular volume do cilindro") // Menu
9       println("2. Sair")
10      val op = StdIn.readInt() // Dando entrada na opção
11
12      op match {
13        case 1 => // Entrada dos dados do cilindro
14          println("Digite o raio do cilindro:")
15          val raio = StdIn.readDouble()
16          println("Digite a altura do cilindro:")
17          val altura = StdIn.readDouble()
18          val volume = calcVol(raio, altura)
19          println(s"O volume do cilindro é: $volume")
20
21        case 2 => // Fim do código
22          continuar = false
23          println("Encerrando o programa.")
24
25        case _ => // Erro na entrada
26          println("Opção inválida. Por favor, tente novamente.")
27      }
28      println()
29    }
30  }
31
32  def calcVol(raio: Double, altura: Double): Double = { // Calculo do volume
33    val areaB = 3.14 * raio * raio
34    val volume = areaB * altura
35    volume // Retorna o volume
36  }
37}

```

Fonte: Autor do Livro

Resultados ao compilar o arquivo:

Figura 4.2: Resultados Implementação Operações

```

1. Calcular volume do cilindro
2. Sair
1
Digite o raio do cilindro:
5
Digite a altura do cilindro:
3
O volume do cilindro é: 235.5

1. Calcular volume do cilindro
2. Sair
2
Encerrando o programa.

```

Fonte: Autor do Livro

4.2 Calculadora

O algoritmo a ser apresentado nesta seção é um exemplo simples de uma calculadora que é capaz de realizar as 4 operações matemáticas básicas: adição, subtração, multiplicação e divisão. O objetivo desse algoritmo é dar a possibilidade ao usuário de realizar as 4 operações com os números fornecidos.

Código fonte:

```

// Definindo a classe Calculadora
class Calculadora {
    // Definindo as operações básicas da classe (
    // métodos)
    def add(a: Int, b: Int): Int = a + b

    def sub(a: Int, b: Int): Int = a - b

    def mult(a: Int, b: Int): Int = a * b

    def div(a: Float, b: Float): Float = {
        if (b != 0) // Verificando se ocorre uma
                    // divisão por 0
            a / b
        else
            throw new ArithmeticException("Divisão
                    por zero não é permitida!")
    }
}

// Definindo o main, onde o programa será rodado
object Main {
    def main(args: Array[String]): Unit = {
        // Criando uma instância da classe
        // Calculadora
        val calculadora = new Calculadora()

        // Realizando as operações
    }
}

```

```
    val sum = calculadora.add(4, 2)
    val dif = calculadora.sub(4, 2)
    val prod = calculadora.mult(4, 2)
    val quot = calculadora.div(4, 2)

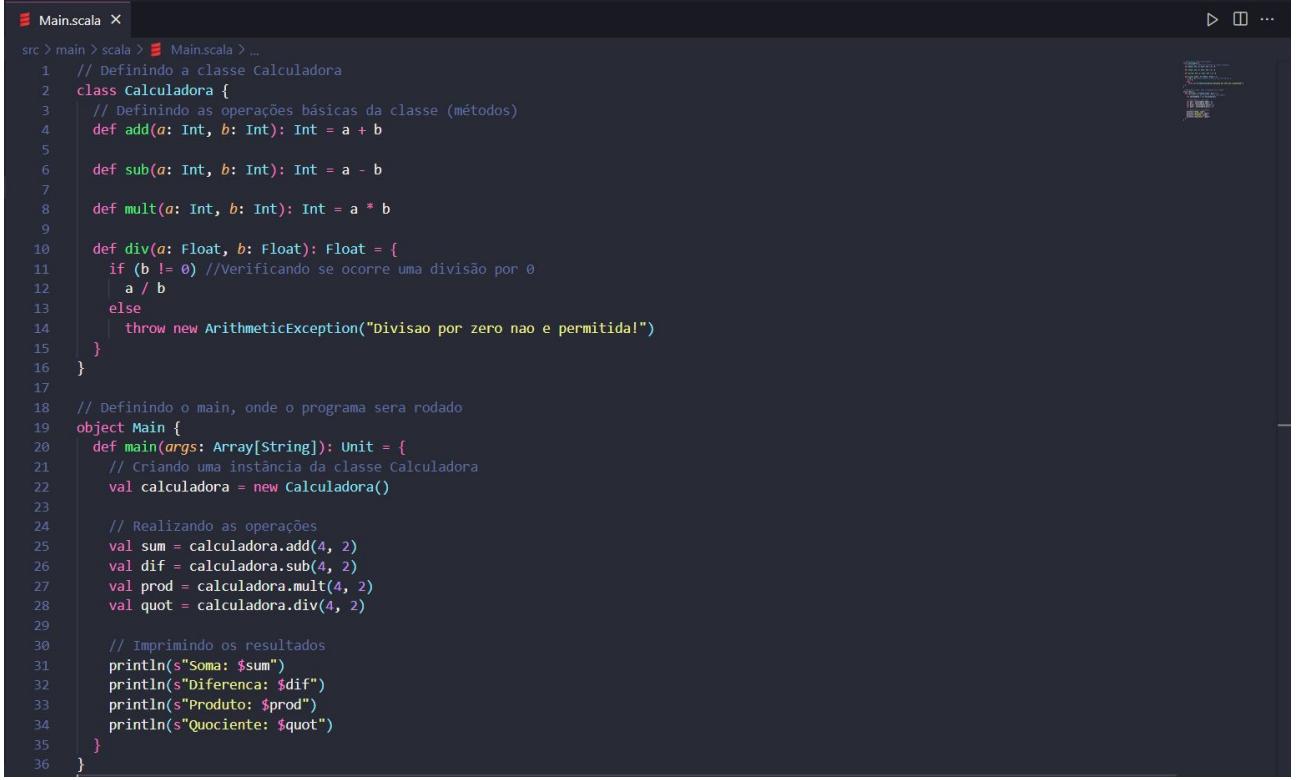
    // Imprimindo os resultados
    println(s"Soma: $sum")
    println(s"Diferenca: $dif")
    println(s"Produto: $prod")
    println(s"Quociente: $quot")
}

}
```

- Inicialmente, é definida uma classe chamada "calculadora", com seus 4 métodos que representam as 4 operações básicas.
- O método "add"recebe 2 valores inteiros e os soma, o método "sub"recebe 2 inteiros e os subtrai, o método "mult"recebe 2 inteiros e os multiplica, e por último o método "div"recebe 2 floats, verifica se o segundo é diferente de 0 (para evitar uma indefinição) e caso não seja será realizada a divisão de a por b.
- Em seguida é criado o main, local que será o ponto de partida do programa.
- No interior do main é criada uma nova instância da classe calculadora.
- Por meio dessa instância criada, são realizadas as operações matemáticas desejadas, armazenando os resultados em variáveis.
- E por último são mostrados os valores para o usuário.

A implementação desse código fonte em uma Ide é extremamente simples, basta criar um projeto e escrever o código em na Ide desejada. Neste caso, foi utilizado como Ide o Vscode, veja o exemplo abaixo:

Figura 4.3: Implementação Calculadora



```

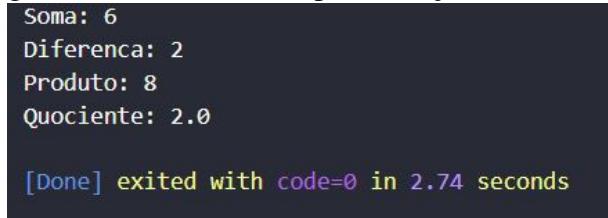
Main.scala x
src > main > scala > Main.scala > ...
1 // Definindo a classe Calculadora
2 class Calculadora {
3     // Definindo as operações básicas da classe (métodos)
4     def add(a: Int, b: Int): Int = a + b
5
6     def sub(a: Int, b: Int): Int = a - b
7
8     def mult(a: Int, b: Int): Int = a * b
9
10    def div(a: Float, b: Float): Float = {
11        if (b != 0) //verificando se ocorre uma divisão por 0
12            a / b
13        else
14            throw new ArithmeticException("Divisao por zero nao e permitida!")
15    }
16 }
17
18 // Definindo o main, onde o programa sera rodado
19 object Main {
20     def main(args: Array[String]): Unit = {
21         // Criando uma instância da classe Calculadora
22         val calculadora = new Calculadora()
23
24         // Realizando as operações
25         val sum = calculadora.add(4, 2)
26         val dif = calculadora.sub(4, 2)
27         val prod = calculadora.mult(4, 2)
28         val quot = calculadora.div(4, 2)
29
30         // Imprimindo os resultados
31         println(s"Soma: $sum")
32         println(s"Diferenca: $dif")
33         println(s"Produto: $prod")
34         println(s"Quociente: $quot")
35     }
36 }

```

Fonte: Autor do Livro

Resultados obtidos ao compilar o algoritmo:

Figura 4.4: Resultados Implementação Calculadora



```

Soma: 6
Diferenca: 2
Produto: 8
Quociente: 2.0

[Done] exited with code=0 in 2.74 seconds

```

Fonte: Autor do Livro

4.3 Regra do Trapézio

Nessa seção será apresentada uma aplicação de um código em Scala responsável por calcular o valor aproximado de uma integral definida em um intervalo de a até b . Para isso será utilizado um conceito do cálculo numéricico denominado Regra do Trapézio Generalizado.

Código fonte:

```

object RegraDoTrapezio {
    def main(args: Array[String]): Unit = {
        val a = 0.0 // Limite inferior do
                    intervalo

```

```

        val b = 2.0 // Limite superior do
                     intervalo
        val n = 10 // Numero de subintervalos

        val h = (b - a) / n // Tamanho de cada
                     subintervalo
        val x = Array.tabulate(n + 1)(i => a + i
            * h) // Pontos xi tal que i = 0, 1,
            ..., n
        val y = x.map(f) // Valores de f(xi) para
                     cada xi

        val integral = (h / 2) * (y.head + 2 * y.
            drop(1).dropRight(1).sum + y.last)

        println(s"Aproximacao da integral:
            $integral")
    }

    def f(x: Double): Double = {
        // Definindo a função que será integrada
        x * x // f(x) = x^2
    }
}

```

- Inicialmente no main é definido os limites inferiores e superiores da integral, assim o número de subintervalos que serão utilizados para o cálculo.
- Com isso será calculado o valor de h , que será o passo entre os pontos de amostragem.
- Após calcular o h (o passo), será efetuado o valor de todos os x_i relacionados a cada subintervalo, assim como o $f(x_i)$ associado a cada subintervalo.
- Com isso será calculado o valor aproximado da integral, para isso basta aplicar a fórmula da regra do trapézio generalizado:

$$\text{Integral} \approx \frac{h}{2} \left(y_0 + 2 \sum_{i=1}^{n-1} y_i + y_n \right)$$

onde:

h : tamanho do subintervalo

y_0, y_1, \dots, y_n : valores da função nos pontos de amostragem

Com isso já é possível implementar esse código em um ambiente de programação adequado. Para isso foi escolhido a Ide Vscode, veja abaixo a implementação do algoritmo e o valor obtido com sua compilação.

Implementação:

Figura 4.5: Implementação Regra do Trapézio Generalizado

```

Main.scala X
src > main > scala > Main.scala > {} RegraDoTrapezio
run | debug
1 object RegraDoTrapezio {
2     def main(args: Array[String]): Unit = {
3         val a = 0.0 // Limite inferior do intervalo
4         val b = 2.0 // Limite superior do intervalo
5         val n = 10 // Número de subintervalos
6
7         val h = (b - a) / n // Tamanho de cada subintervalo
8         val x = Array.tabulate(n + 1)(i => a + i * h) // Pontos xi tal que i = 0, 1, ..., n
9         val y = x.map(f) // Valores de f(xi) para cada xi
10
11        val integral = (h / 2) * (y.head + 2 * y.drop(1).dropRight(1).sum + y.last)
12
13        println(s"Aproximacao da integral: $integral")
14    }
15
16    def f(x: Double): Double = {
17        // Definindo a função que será integrada
18        x * x // f(x) = x^2
19    }
20}
21

```

Fonte: Autor do Livro

Resultado obtido ao compilar o algoritmo:

Figura 4.6: Resultados Implementação Regra do Trapézio Generalizado

```

Aproximacao da integral: 2.6800000000000006
[Done] exited with code=0 in 2.938 seconds

```

Fonte: Autor do Livro

4.4 Bubble Sort

O bubble sort é um clássico algoritmo de ordenação de dados muito utilizado para organizar valores de um vetor. O algoritmo funciona percorrendo todos os elementos de uma lista, comparando os valores adjacentes e caso estejam na ordem errada é realizada uma troca. Isso ocorre até o array estar completamente ordenado.

Código fonte:

```

object BubbleSortExample {
    def BubbleSort(arr: Array[Int]): Array[Int] = {
        // Função de bubble sort
        val n = arr.length // Definindo o tamanho do array
    }
}
```

```

        for (i <- 0 until n - 1) { // For para
            percorrer o array
            for (j <- 0 until n - i - 1) { //
                For para percorrer o array e
                realizar as comparacoes
                if (arr(j) > arr(j + 1))
                    { // Verificando se o
                        proximo valor e menor
                        que o valor atual
                        val temp = arr(j)
                        // Troca
                        arr(j) = arr(j +
                            1)
                        arr(j + 1) = temp
                }
            }
            arr // Retorna
        }

def main(args: Array[String]): Unit = {
    Definindo o main
    val array = Array(64, 34, 25, 12, 22, 11,
        90) // Definindo o array
    val sortArray = BubbleSort(array) //
        Chamada da funcao
    println(sortArray.mkString(", ")) //
        Mostrando para o usuario
}
}

```

- Inicialmente no main, é definido um array com elementos aleatórios de maneira desordenada.
- Em seguida é realizada a chamada da função bubble sort.
- Logo em seguida, o tamanho do array é salvo na variável.
- Em seguida é criado um for externo responsável por percorrer o array.
- Também é criado um for interno, responsável por percorrer o array e realizar as comparações entre o elemento na posição j e seu sucessor, caso o valor na posição j seja maior que na posição j + 1, ocorre uma troca.
- Com o fim de ambos os loops, o array organizado é retornado e mostrado ao usuário.

A implementação desse algoritmo, assim como as anteriores, foi feita criando um projeto na Ide Vscode, veja abaixo a implementação e o resultado obtido com a compilação do arquivo.

Implementação:

Figura 4.7: Implementação Bubble Sort



```

Main.scala X
src > main > scala > Main.scala > ...
● 1 ✓ object BubbleSortExample {
  2   def BubbleSort(arr: Array[Int]): Array[Int] = { // Funcao de bubble sort
  3     val n = arr.length // Definindo o tamanho do array
  4     for (i <- 0 until n - 1) { // For para percorrer o array
  5       for (j <- 0 until n - i - 1) { // For para percorrer o array e realizar as comparacoes
  6         if (arr(j) > arr(j + 1)) { // Verificando se o proximo valor e menor que o valor atual
  7           val temp = arr(j) // Troca
  8           arr(j) = arr(j + 1)
  9           arr(j + 1) = temp
 10        }
 11      }
 12    }
 13    arr // Retorna
 14  }
 15
 16  def main(args: Array[String]): Unit = { // Definindo o main
 17    val array = Array(64, 34, 25, 12, 22, 11, 90) // Definindo o array
 18    val sortArray = BubbleSort(array) // Chamada da funcao
 19    println(sortArray.mkString(", "))
 20  }
 21}

```

Fonte: [Clique Aqui](#)

Resultados obtidos quando o arquivo é compilado:

Figura 4.8: Resultados Implementação Bubble Sort

```

11, 12, 22, 25, 34, 64, 90
[Done] exited with code=0 in 2.96 seconds

```

Fonte: [Clique Aqui](#)

4.5 Quick Sort

O quick sort é um clássico e extremamente efetivo algoritmo de ordenação, por isso recebe o nome de quick sort. Ele funciona selecionando um elemento do array que será chamado de pivô, com isso é seguido a lógica de separar o vetor em 2, uma parte com valores menores que o pivô e outra parte com valores maiores que o pivô, buscando assim coloca-lo em sua respectiva posição.

Código fonte:

```

object QuickSort {
  def main(args: Array[String]): Unit = { // Main
    val arr = Array(64, 34, 25, 12, 22, 11,
                  90) // Definindo array
    println("Array antes da ordenacao:")
    println(arr.mkString(", "))
  }
}

```

```
        quickSort(arr, 0, arr.length - 1) //  
            Chamada da funcao  
  
        println("Array apos a ordenacao:")  
        println(arr.mkString(", "))  
    }  
  
    def quickSort(arr: Array[Int], a: Int, b: Int):  
        Unit = { // Definindo funcao quick sort  
            if (a < b) { // Caso ainda tenham dados  
                no array  
                val pivo = partition(arr, a, b)  
  
                // Chamadas recursivas  
                quickSort(arr, a, pivo - 1)  
                quickSort(arr, pivo + 1, b)  
            }  
        }  
  
    def partition(arr: Array[Int], a: Int, b: Int):  
        Int = {  
            val pivo = arr(b) // Define o pivo  
            var i = a - 1  
  
            for (j <- a until b) { // Loop para  
                percorrer o array  
                if (arr(j) <= pivo) { //  
                    Procurando menor valor  
                    i += 1  
                    swap(arr, i, j) // Funcao  
                    de troca  
                }  
            }  
  
            swap(arr, i + 1, b) // Funcao de troca  
            i + 1  
        }  
  
    def swap(arr: Array[Int], i: Int, j: Int): Unit =  
    { // Funcao de troca  
        val temp = arr(i)  
        arr(i) = arr(j)  
        arr(j) = temp  
    }  
}
```

- Inicialmente é criado o main e dentro dele é também criado o array com valores aleatórios de maneira desordenada, em seguida é realizada a chamada da função, passando como parâmetros o próprio array, o início e o fim.
- Após isso é verificado se ainda existem elementos no vetor, se verdadeiro é chamada a função que repartira o array e selecionará o pivô.
- Dentro dessa função é definido o pivô como o último elemento do array, dentro do loop é verificado se o elemento for menor ou igual ao pivô, incrementamos i e trocamos o elemento na posição i com o elemento na posição j.
- Após o loop, o último elemento menor ou igual ao pivô está na posição i + 1. Portanto, trocamos o pivô de posição com esse elemento para colocá-lo em sua posição final.
- Com o fim do loop é retornado a posição do pivô, com isso são feitas as chamadas recursivas dividindo novamente o array, sendo a primeira chamada feita para organizar a esquerda a lista, e a segunda para ordenar a direita do array.

A implementação desse algoritmo, assim como as anteriores, foi feita criando um projeto na Ide Vscode, veja abaixo a implementação e o resultado obtido com a compilação do arquivo.

Implementação:

Figura 4.9: Implementação Quick Sort Parte 1

```

Main.scala ×
src > main > scala > Main.scala > {} QuickSort > swap
1 √ object QuickSort {
2 √   def main(args: Array[String]): Unit = { // Main
3     val arr = Array(64, 34, 25, 12, 22, 11, 90) // Definindo array
4     println("Array antes da ordenacao:")
5     println(arr.mkString(", "))
6
7     quickSort(arr, 0, arr.length - 1) // Chamada da funcao
8
9     println("Array apos a ordenacao:")
10    println(arr.mkString(", "))
11  }
12
13 √ def quickSort(arr: Array[Int], a: Int, b: Int): Unit = { // Definindo funcao quick sort
14   if (a < b) { // Caso ainda tenham dados no array
15     val pivo = partition(arr, a, b)
16
17     // Chamadas recursivas
18     quickSort(arr, a, pivo - 1)
19     quickSort(arr, pivo + 1, b)
20   }
21 }
22
23 √ def partition(arr: Array[Int], a: Int, b: Int): Int = {
24   val pivo = arr(b) // Define o pivo
25   var i = a - 1
26
27   √ for (j <- a until b) { // Loop para percorrer o array
28     if (arr(j) <= pivo) { // Procurando menor valor
29       i += 1
30       swap(arr, i, j) // Funcao de troca
31     }
32   }
33
34   swap(arr, i + 1, b) // Funcao de troca
35   i + 1
36 }
37

```

Fonte: [Clique Aqui!](#)

Figura 4.10: Implementação Quick Sort Parte 2

```

37
38   def swap(arr: Array[Int], i: Int, j: Int): Unit = { // Funcao de troca
39     val temp = arr(i)
40     arr(i) = arr(j)
41     arr(j) = temp
42   }
43 }

```

Fonte: [Clique Aqui!](#)

Resultados obtidos ao compilar:

Figura 4.11: Resultados Implementação Quick Sort

```
Array antes da ordenacao:  
64, 34, 25, 12, 22, 11, 90  
Array apos a ordenacao:  
11, 12, 22, 25, 34, 64, 90  
  
[Done] exited with code=0 in 3.609 seconds
```

Fonte: [Clique Aqui!](#)

4.6 Equação do Segundo Grau

Essa seção abordará um algoritmo básico de cálculo de uma equação do segundo grau utilizando a conhecida fórmula de bhaskhara. O objetivo desse algoritmo é encontrar as raízes de uma equação do segundo grau fornecida ao programa.

```
import scala.math.sqrt // Biblioteca para calculo de raiz

object EquacaoSegundoGrau {
    def CalcEq(a: Double, b: Double, c: Double): Unit
    = {
        val delta = b * b - 4 * a * c // Calculo
        delta

        if (delta > 0) { // Existem 2 raizes
            reais
                val x1 = (-b + sqrt(delta)) / (2
                    * a) // Calculo raiz 1
                val x2 = (-b - sqrt(delta)) / (2
                    * a) // Calculo raiz 2
                println("Raizes:")
                println(s"x1 = $x1")
                println(s"x2 = $x2")
        } else if (delta == 0) { // Existe 1 raiz
            val x = -b / (2 * a) // Calculo
            da raiz
            println("Raiz:")
            println(s"x = $x")
        } else {
            println("A equacao nao possui
            raizes reais.") // Delta < 0,
            logo raiz imaginaria
        }
    }

    def main(args: Array[String]): Unit = { // Main
        // Coeficientes equacao
```

```

        val a = 1.0
        val b = -3.0
        val c = 2.0

    CalcEq(a, b, c) // Chamada funcao
}

}

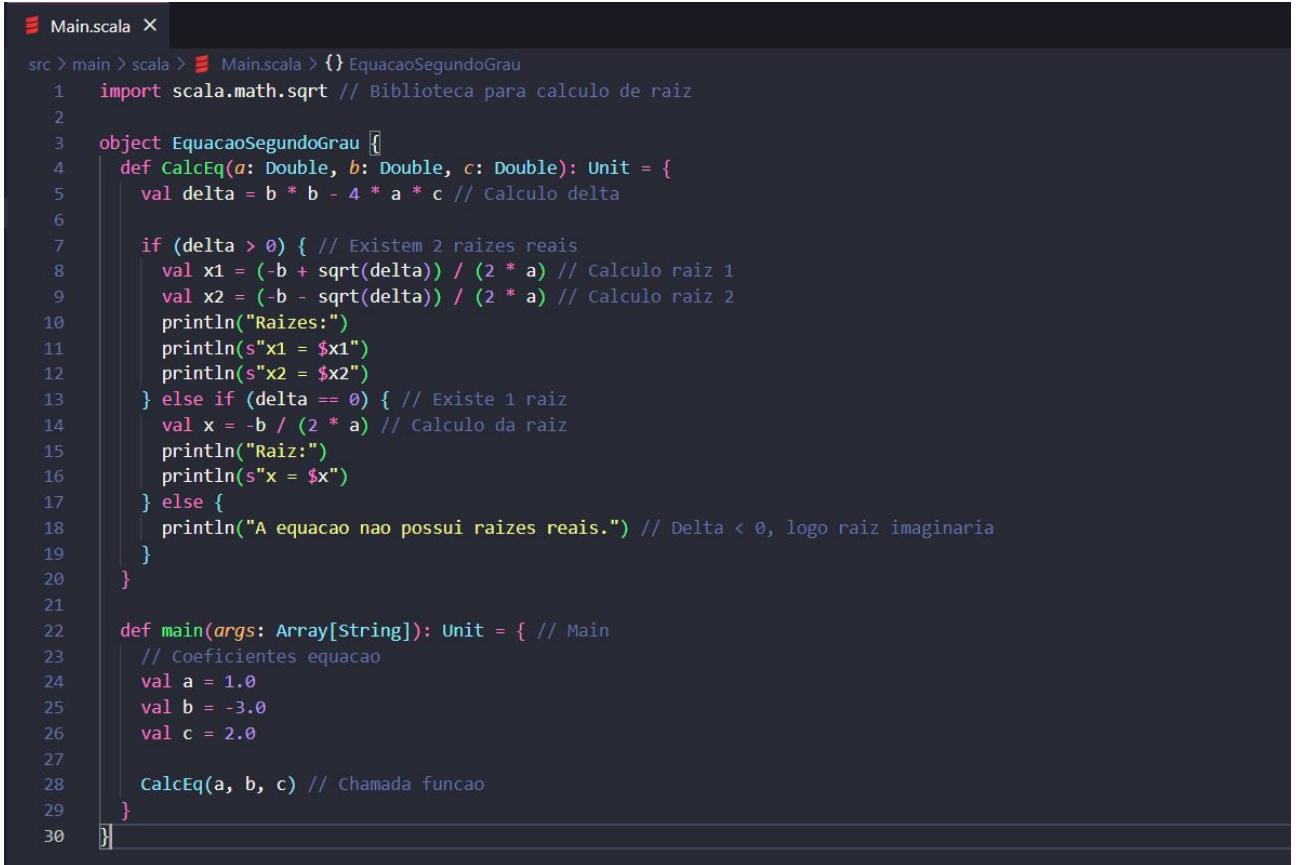
```

- Inicialmente é dado o import na biblioteca responsável por calcular o valor da raiz quadrada.
- no main é dada a entrada dos coeficientes da equação do segundo grau.
- Na função chamada, é calculado inicialmente o delta, em seguida é verificado se o mesmo é maior que 0, se sim existem 2 raízes que serão calculada e retornadas, se for zero existe apenas uma raiz, e se for menor que 0 ocorrerá uma raiz quadrada de número negativo, logo uma raiz imaginária.
- Em seguida o valor é mostrado ao usuário.

A implementação desse código fonte em uma Ide é extremamente simples, basta criar um projeto e escrever o código em na Ide desejada. Neste caso, foi utilizado como Ide o Vscode, veja o exemplo abaixo:

Implementação:

Figura 4.12: Implementação Equação 2 Grau



```

Main.scala X
src > main > scala > Main.scala > {} EquacaoSegundoGrau
1   import scala.math.sqrt // Biblioteca para calculo de raiz
2
3   object EquacaoSegundoGrau {
4       def CalcEq(a: Double, b: Double, c: Double): Unit = {
5           val delta = b * b - 4 * a * c // Calculo delta
6
7           if (delta > 0) { // Existem 2 raizes reais
8               val x1 = (-b + sqrt(delta)) / (2 * a) // Calculo raiz 1
9               val x2 = (-b - sqrt(delta)) / (2 * a) // Calculo raiz 2
10              println("Raizes:")
11              println(s"x1 = $x1")
12              println(s"x2 = $x2")
13          } else if (delta == 0) { // Existe 1 raiz
14              val x = -b / (2 * a) // Calculo da raiz
15              println("Raiz:")
16              println(s"x = $x")
17          } else {
18              println("A equacao nao possui raizes reais.") // Delta < 0, logo raiz imaginaria
19          }
20      }
21
22      def main(args: Array[String]): Unit = { // Main
23          // Coeficientes equacao
24          val a = 1.0
25          val b = -3.0
26          val c = 2.0
27
28          CalcEq(a, b, c) // Chamada funcao
29      }
30  }

```

Fonte: Autor do Livro

Resultados da compilação:

Figura 4.13: Resultados Equação do 2 Grau

```
Raizes:  
x1 = 2.0  
x2 = 1.0  
  
[Done] exited with code=0 in 2.631 seconds
```

Fonte: Autor do Livro

4.7 Sistema Linear

Nesta seção abordaremos um algoritmo capaz de solucionar um sistema de equações 2x2. Para isso é necessário calcular o determinante da matriz com o objetivo de verificar se existe solução, e caso esse determinante seja diferente de 0, será efetuada a regra de cramer para calcular a solução do sistema.

Código fonte:

```
object SistemaEquacoes {  
  
    def resSist(a11: Double, a12: Double, b1: Double,  
               a21: Double, a22: Double, b2: Double): Option  
        [(Double, Double)] = {  
        val det = a11 * a22 - a12 * a21 //  
            Calculo do determinante  
  
        if (det != 0) {  
            // Calculo solucao  
            val x = (b1 * a22 - b2 * a12) /  
                det  
            val y = (a11 * b2 - a21 * b1) /  
                det  
            Some(x, y)  
        } else {  
            None // O sistema nao tem solucao  
                  unica  
        }  
    }  
  
    def main(args: Array[String]): Unit = {  
        // Coeficientes  
        val a11 = 2.0  
        val a12 = 3.0  
        val b1 = 10.0  
        val a21 = 1.0  
        val a22 = -1.0  
        val b2 = -5.0
```

```

    val solucao = resSist(a11, a12, b1, a21,
                          a22, b2) // Chamada função

    solucao match {
      case Some((x, y)) =>
        println(s"Solução: x = $x, y = $y")
      case None =>
        println("O sistema não tem
               solução única...")
    }
}

```

- Inicialmente no main é se define o sistema linear desejado no seguinte formato:

$$\begin{aligned} a_{11} * x + a_{12} * y &= b_1 \\ a_{21} * x + a_{22} * y &= b_2 \end{aligned}$$

Com isso é realizada a chamada da função que irá calcular a solução.

- Na função de cálculo de solução é inicialmente calculado o determinante, para verificar se o sistema em questão possui solução ou não, caso o determinante seja diferente de zero é realizada a regra de cramer que dará como resultado a solução do sistema, retornando assim para o usuário. Porém caso o determinante seja igual a 0 será retornado 'none', indicando que o sistema não possui solução única.

Com isso já é possível implementar esse código em um ambiente de programação adequado. Para isso foi escolhido a Ide Vscode, veja abaixo a implementação do algoritmo e o valor obtido com sua compilação.

Implementação:

Figura 4.14: Implementação Sistema Linear

```

build.sbt      Main.scala ×
src > main > scala > Main.scala > {} SistemaEquacoes
run | debug
1  object SistemaEquacoes {
2
3    def ressist(a11: Double, a12: Double, b1: Double, a21: Double, a22: Double, b2: Double): Option[(Double, Double)] = {
4      val det = a11 * a22 - a12 * a21 // Calculo do determinante
5
6      if (det != 0) {
7        // Calculo solucao
8        val x = (b1 * a22 - b2 * a12) / det
9        val y = (a11 * b2 - a21 * b1) / det
10       Some(x, y)
11     } else {
12       None // O sistema nao tem solucao unica
13     }
14   }
15
16  def main(args: Array[String]): Unit = {
17    // Coeficientes
18    val a11 = 2.0
19    val a12 = 3.0
20    val b1 = 10.0
21    val a21 = 1.0
22    val a22 = -1.0
23    val b2 = -5.0
24
25    val solucao = ressist(a11, a12, b1, a21, a22, b2) // Chamada funcao
26
27    solucao match {
28      case Some(x, y) =>
29        println(s"Solucao: x = $x, y = $y")
30      case None =>
31        println("O sistema nao tem solucao unica...")
32    }
33  }
34}
35

```

Fonte: Autor do Livro

Resultados da compilação:

Figura 4.15: Resultados Sistema Linear

```

Solucao: x = -1.0, y = 4.0
[Done] exited with code=0 in 2.593 seconds

```

Fonte: Autor do Livro

5. Ferramentas existentes e utilizadas

Neste capítulo serão apresentadas duas ferramentas que foram utilizadas no desenvolvimento desse livro, e que também podem ser usadas pelo leitor. São elas: o Visual Studio Code (VS Code) e o Scastie.

- Nome da ferramenta (compilador-interpretador)
- Endereço na Internet
- Versão atual e utilizada
- Descrição simples (máx 2 parágrafos)
- Telas capturadas da ferramenta
- Outras informações

5.1 Visual Studio Code (VS Code)

Segundo [Pla20], o Visual Studio Code, ou simplesmente VS Code, é uma IDE desenvolvida pela Microsoft. Ela é um ambiente de programação gratuito e de código aberto que possui o suporte para inúmeras linguagens de programação diferentes e que pode ser executada em diversos sistemas operacionais, como o Windows, macOS ou Linux.

Uma das principais características do VS Code é apresentar uma interface gráfica extremamente leve e personalizável, além disso a IDE fornece ao usuário uma diversa quantia de recursos como extensões que podem ser instaladas fornecendo ao usuário uma ampla gama de possibilidades. Isso faz com que a IDE suporte várias linguagens de programação diferentes, e para que a IDE rode códigos em Scala, é necessário duas extensões diferentes, a primeira o Scala Syntax e o Scala Metals, e além disso é necessário compilar os códigos na linguagem, para isso deve-se instalar a extensão Code runner. Vale lembrar que durante o desenvolvimento desse livro, foi utilizada a IDE na versão 1.79.1.

A IDE pode ser instalada em sua atual versão [aqui](#).

Imagens da ferramenta com a implementação de alguns códigos simples:

```

File Edit Selection View Go Run Terminal Help
EXPLORER ...
src > main > scala > Main.scala > {} BubbleSortExample
1 object BubbleSortExample {
2     def bubblesort(arr: Array[Int]): Array[Int] = { // Funcao de bubble sort
3         val n = arr.length // Definindo o tamanho do array
4         for (i <- 0 until n - 1) { // For para percorrer o array
5             for (j <- 0 until n - i) { // For para percorrer o array e realizar as comparacoes
6                 if (arr(j) > arr(j + 1)) { // Verificando se o proximo valor e menor que o valor atual
7                     val temp = arr(j) // Troca
8                     arr(j) = arr(j + 1)
9                     arr(j + 1) = temp
10                }
11            }
12        }
13    arr // Retorna
14 }
15
16 def main(args: Array[String]): Unit = { // Definindo o main
17     val array = Array(64, 34, 25, 12, 22, 11, 98) // Definindo o array
18     val sortArray = bubblesort(array) // Chamada da funcao
19     println(sortArray.mkString(", ")) // Mostrando para o usuario
20 }
21

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

[Running] scala "c:\Users\Gabriel\Desktop\Costa\Estudios\Scala\bubble-sort\src\main\scala>Main.scala"
11, 12, 22, 25, 34, 64, 98

[Done] exited with code=0 in 4.847 seconds

Ln 21, Col 2 Spaces: 2 UTF-8 CRLF Scala Go Live

Figura 5.1:

```

File Edit Selection View Go Run Terminal Help
EXPLORER ...
src > main > scala > Main.scala > {} Fatorial > main
1 object Fatorial {
2
3     def calcFat(num: Int): Int = {
4         if (num == 0) // Caso base recursao
5             1
6         else
7             num * calcFat(num - 1) // Chamada Recursiva
8     }
9
10    def main(args: Array[String]): Unit = {
11        val num = 5 // Numero que deseja obter o fatorial
12        val res = calcFat(num) // Chamada da funcao
13        println(s"$0! factorial de $num e: $res")
14    }
15

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Ln 13, Col 26 Spaces: 2 UTF-8 CRLF Scala Go Live

Figura 5.2:

The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Terminal:** Main.scala - equacao-segundo-grau - Visual Studio Code
- Explorer:** Shows a project structure for "EQUACAO-SEGUNDO-GRAU" containing .metals, .vscode, project, src/main.scala (with Main.scala selected), and build.sbt.
- Code Editor:** Displays the following Scala code for calculating quadratic roots:

```

src/main/scala/Main.scala
1 import scala.math.sqrt // Biblioteca para calculo de raiz
2
3 object EquacaoSegundoGrau {
4     def CalcEq(a: Double, b: Double, c: Double): Unit = {
5         val delta = b * b - 4 * a * c // calculo delta
6
7         if (delta > 0) { // Existem 2 raizes reais
8             val x1 = (-b + sqrt(delta)) / (2 * a) // calculo raiz 1
9             val x2 = (-b - sqrt(delta)) / (2 * a) // calculo raiz 2
10            println("Raizes:")
11            println(s"x1 = $x1")
12            println(s"x2 = $x2")
13        } else if (delta == 0) { // Existe 1 raiz
14            val x = -b / (2 * a) // Calculo da raiz
15            println("Raiz:")
16            println(s"x = $x")
17        } else {
18            println("A equacao nao possui raizes reais.") // Delta < 0, logo raiz imaginaria
19        }
20    }
21
22    def main(args: Array[String]): Unit = { // Main
23        // Coeficientes equacao
24        val a = 1.0
25        val b = -3.0
26        val c = 2.0
27
28        CalcEq(a, b, c) // Chamada funcao
29    }
}

```

- Bottom Status Bar:** Ln 30, Col 2, Spaces: 2, UTF-8, CRLF, Scala, Go Live, etc.

Figura 5.3:

Imagen da aba de extensões do VS Code:

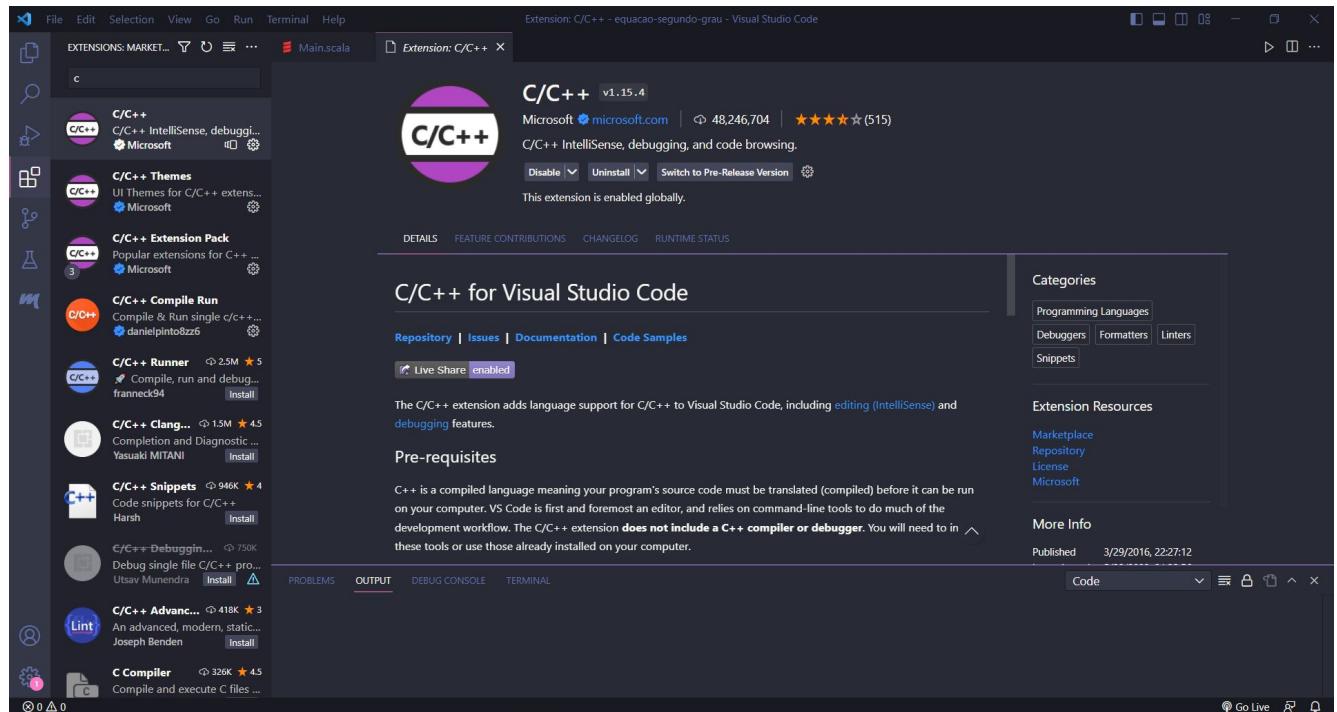


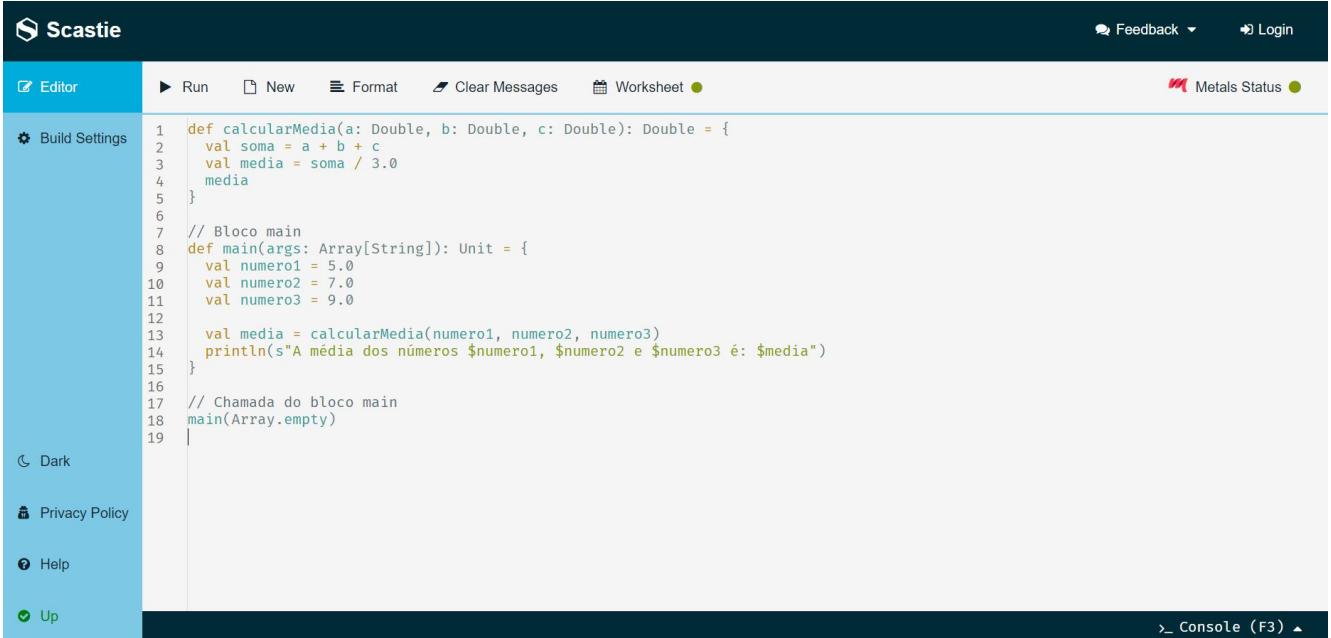
Figura 5.4:

5.2 Scastie

O Scastie é uma plataforma online que permite executar código em Scala. É um ambiente de programação baseado na web criado com o intuito de ser de fácil acesso e interativo, e com o principal propósito: não precisar instalar nenhum software.

Uma das características mais notáveis do Scastie é a capacidade de compartilhar códigos com outros usuários, facilitando a colaboração de diferentes programadores. Além disso, o ambiente de programação possibilita o usuário importar bibliotecas. Isso faz com que o Scastie seja uma ótima ferramenta para pessoas que desejam estudar, testar e compartilhar códigos em Scala.

A ferramenta pode ser encontrada no próprio site do Scala ou clicando [aqui](#).



The screenshot shows the Scastie web application. At the top, there's a dark header bar with the Scastie logo, a feedback link, and a login link. Below the header is a toolbar with buttons for Run, New, Format, Clear Messages, and Worksheet. On the left side, there's a sidebar with a 'Build Settings' section containing a 'Dark' theme option. The main area is a code editor with the following Scala code:

```

1 def calcularMedia(a: Double, b: Double, c: Double): Double = {
2   val soma = a + b + c
3   val media = soma / 3.0
4   media
5 }
6
7 // Bloco main
8 def main(args: Array[String]): Unit = {
9   val numero1 = 5.0
10  val numero2 = 7.0
11  val numero3 = 9.0
12
13  val media = calcularMedia(numero1, numero2, numero3)
14  println(s"A média dos números $numero1, $numero2 e $numero3 é: $media")
15
16 // Chamada do bloco main
17 main(Array.empty)
18
19

```

To the right of the code editor is a large white area representing the 'Worksheet' where the code would run. At the bottom right of the main area, there's a 'Console (F3)' button.

Figura 5.5:

The screenshot shows the Scastie web interface. On the left is a sidebar with links for Editor, Build Settings, Dark mode, Privacy Policy, Help, and Up. The main area contains the following Scala code:

```
object SomaProduto {
  def main(args: Array[String]): Unit = {
    val numero1 = 5
    val numero2 = 3

    val soma = somar(numero1, numero2)
    val produto = multiplicar(numero1, numero2)

    println("A soma dos números é: " + soma)
    println("O produto dos números é: " + produto)
  }

  def somar(a: Int, b: Int): Int = {
    a + b
  }

  def multiplicar(a: Int, b: Int): Int = {
    a * b
  }
}
```

At the bottom right of the main area, there is a link labeled "Console (F3) ▲".

Figura 5.6:

The screenshot shows the Scastie web interface. On the left is a sidebar with links for Editor, Build Settings, Dark mode, Privacy Policy, Help, and Up. The main area contains the following Scala code:

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

At the bottom right of the main area, there is a link labeled "Console (F3) ▲".

Figura 5.7:



6. Considerações Finais

Ao longo desse livro foram abordados diversos aspectos sobre a linguagem de programação Scala. Inicialmente foi apresentada uma visão geral sobre a linguagem, seguido pela sua história que apesar de ser recente, não existem muitos relatos disponíveis para o público, e por último foram apresentadas algumas aplicações da linguagem no mercado de trabalho.

Após esse capítulo introdutório, foi abordado durante alguns capítulos aspectos básicos e avançados necessários para a programação em Scala. Em seguida foram demonstradas algumas aplicações utilizando esses conceitos apresentados anteriormente.

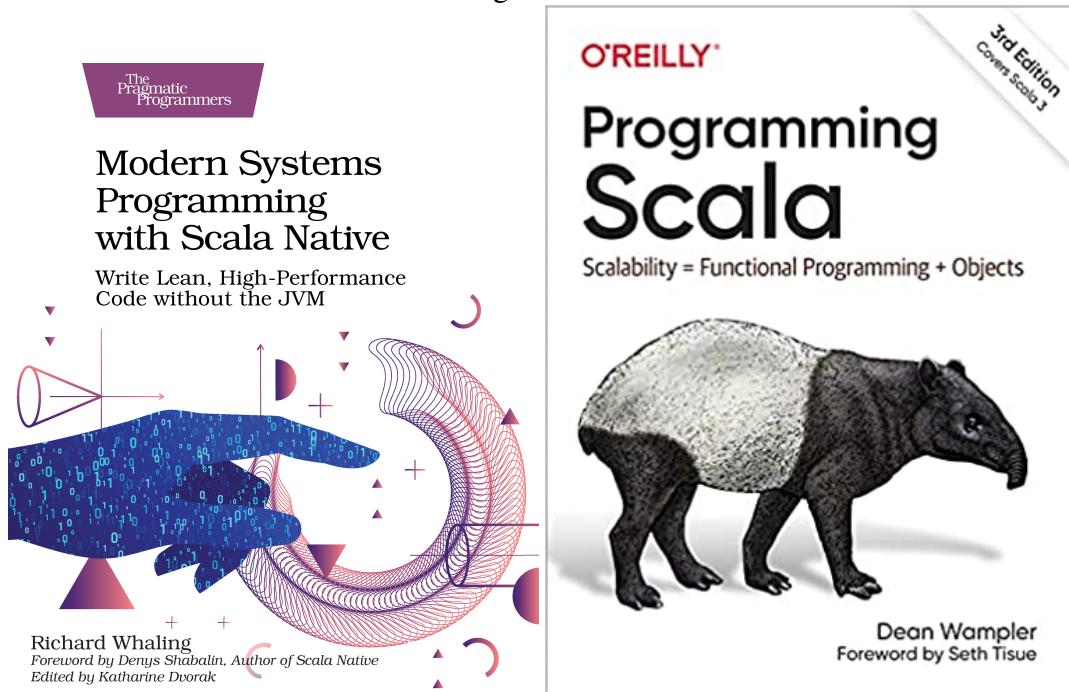
É válido lembrar que esse livro foi escrito com o intuito de trazer uma introdução ao Scala junto com alguns necessários para a programação básica na linguagem, com o principal intuito de promover ao leitor o interesse de se aprofundar no estudo da linguagem Scala. Aspectos do Scala que podem ser estudados futuramente:

- Traits e Mixins: Estudar o uso de traits com o objetivo de compartilhar os comportamentos entre classes e como compor traits pode ser algo vantajoso em muitas situações adversas.
- Programação Assíncrona: Estudar as bibliotecas Promises, Akka e Futures para poder utilizar paralelismo e concorrência de maneira efetiva pode garantir uma ótima eficiência do código.
- Macros: Estudar a utilização de macros para criar o código em tempo de compilação pode garantir muitas vantagens para o usuário.
- Implicits: Estudar o uso de implicits com o objetivo de adicionar comportamentos ou tipos implicitamente pode facilitar a utilização de inúmeras bibliotecas.
- Concorrência e Paralelismo: Como já citado antes, estudar o uso de paralelismo e a concorrência pode aumentar muito a eficiência do código em Scala.

Portanto, é possível concluir que o Scala é uma linguagem de programação extremamente versátil e útil e diversas áreas diferentes, tanto para o mercado de trabalho quanto para estudo. Isso, junto de sua similaridade com Java faz com que sua utilização tenha aumentado cada vez mais entre os programadores. Por isso, caso o leitor tenha o interesse

em continuar ou reforçar os estudos na linguagem, são recomendados os seguintes livros:

Figura 6.1:



Fonte: Autores

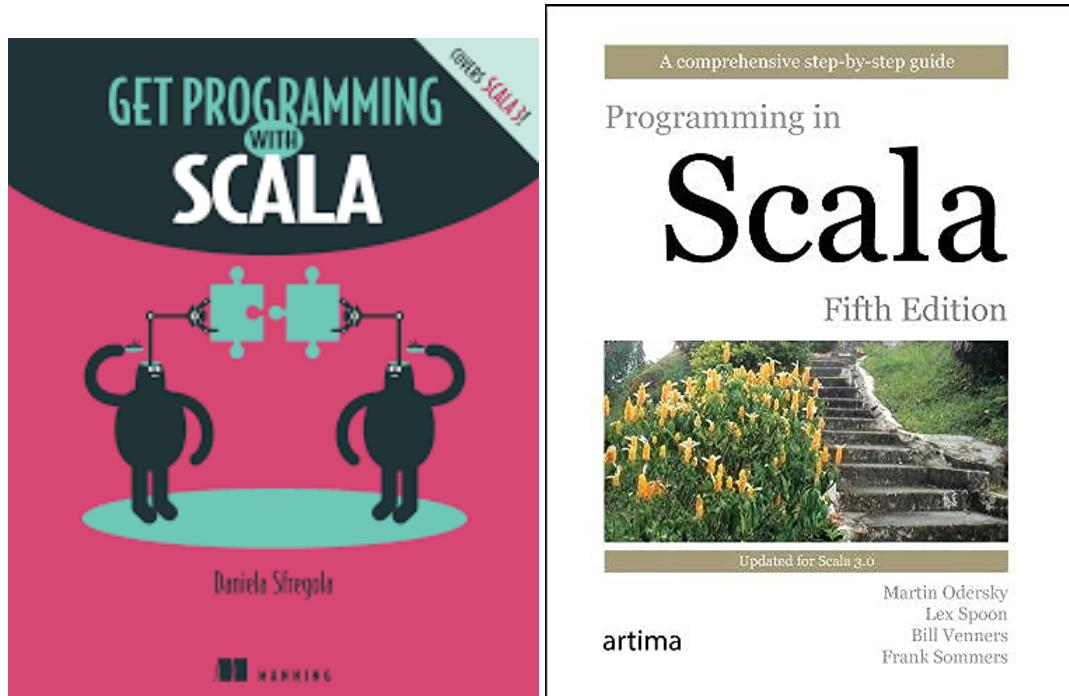


Figura 6.2:

Fonte: Autores

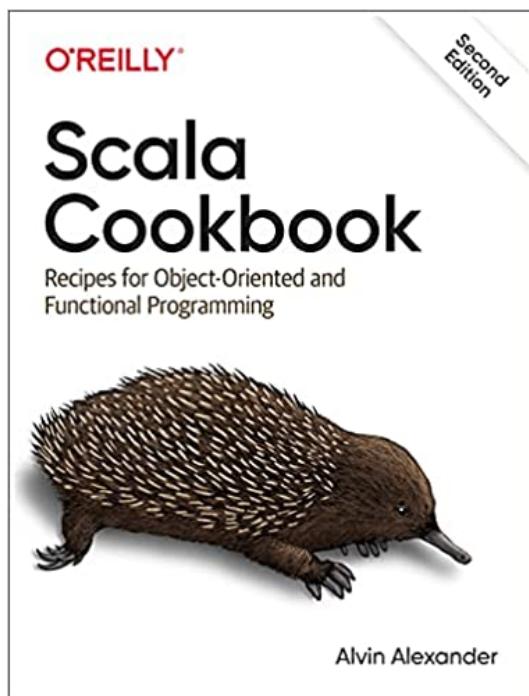


Figura 6.3:
Fonte: Autores



Referências Bibliográficas

- [OMS^{+21]} Odersky, Martin, Spoon, Lex, Venners, Bill, Sommers, and Frank. *Programming in Scala Fifth Edition*. Artima Press, Walnut Creek, CA, June 15, 2021. Citado 10 vezes nas páginas 7, 15, 21, 22, 24, 27, 28, 31, 32 e 35.
- [Pla20] Michael Plainer. Study of visual studio code, practical course — contributing to an open-source project. page 10, 2020. Citado na página 55.
- [Sfr21] Daniela Sfregola. *Get Programming with Scala*. Manning Publications Co. LLC, Shelter Island, NY, 2021. Citado 8 vezes nas páginas 7, 9, 10, 15, 23, 27, 28 e 29.
- [VS09] Bill Venners and Frank Sommers. The origins of scala, a conversation with martin odersky, part i. 2009. Citado 2 vezes nas páginas 8 e 9.
- [Wam21] Dean Wampler. *Programming Scala Scalability = Functional Programming + Objects*. O'Reilly Media, Incorporated, 1005 Gravenstein Highway North, Sebastopol, CA, 2021. Citado 6 vezes nas páginas 7, 15, 19, 23, 25 e 27.

