



Introdução à Linguagem Scala

Paradigmas de Linguagens de Programação

Marcos Bruno P. Campos

July 10, 2024

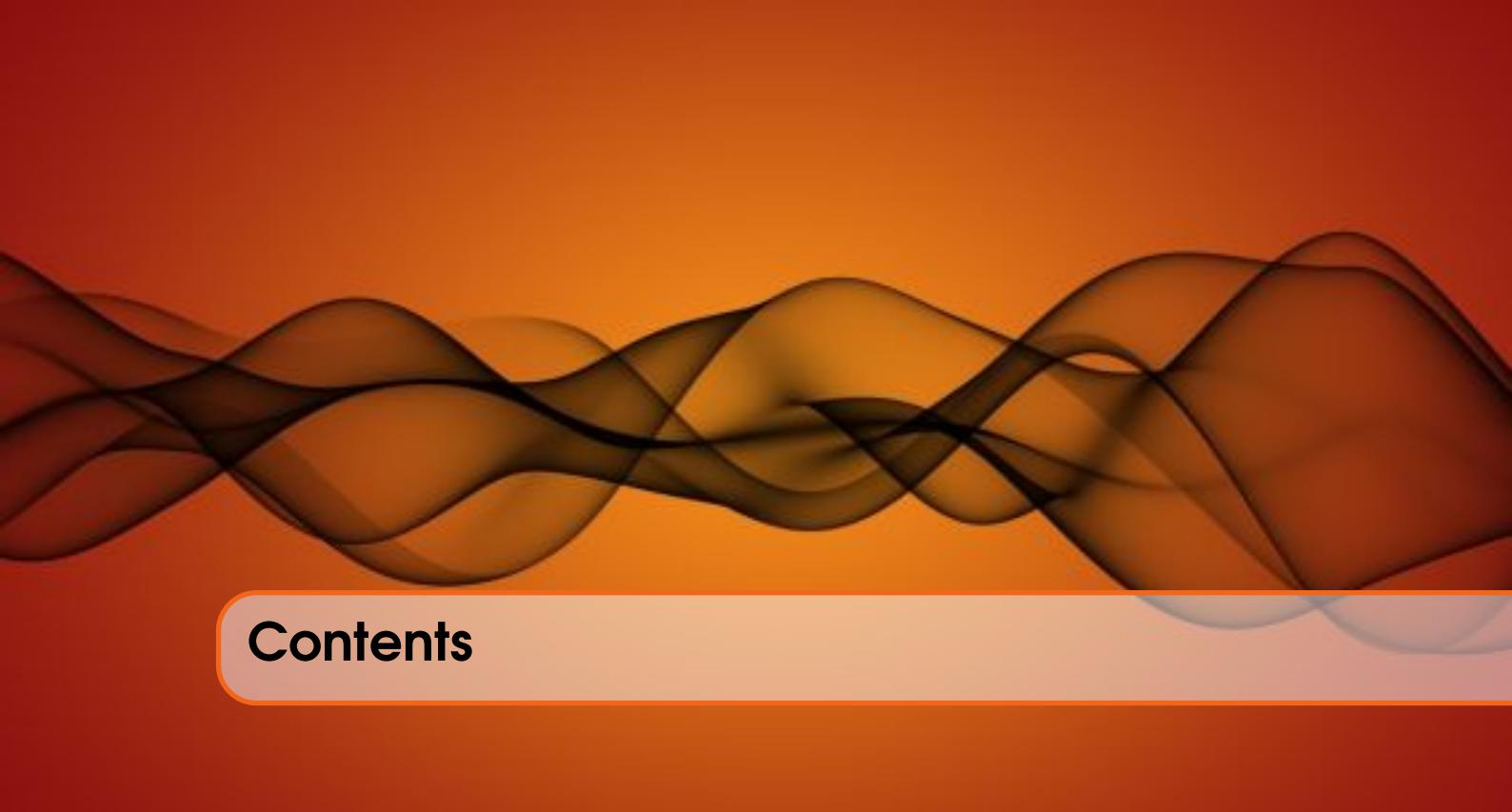


Copyright © 2024 Marcos Bruno P. Campos e Ausberto S. Castro Vera

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



Contents

1	Introdução	5
1.1	Aspectos históricos da linguagem Scala	5
1.2	Áreas de Aplicação da Linguagem	6
1.2.1	Desenvolvimento Web	6
1.2.2	Processamento de Big Data	6
1.2.3	Machine Learning e Inteligência Artificial	6
1.2.4	Desenvolvimento de Aplicações Desktop e Mobile	7
1.2.5	Finanças e Serviços Financeiros	7
1.2.6	Educação e Pesquisa	7
2	Conceitos básicos da Linguagem Scala	9
2.1	Variáveis e constantes	9
2.2	Tipos de Dados	9
2.2.1	Tipo de Dados Primitivos	10
2.2.2	Tipos de Dados Compostos	12
2.3	Operadores em Scala	13
2.3.1	Operadores Aritméticos	13
2.3.2	Operadores de Comparação	13
2.3.3	Operadores Lógicos	14
3	Programação em Scala	15
3.1	Entradas e Saídas	15
3.1.1	Entrada	15
3.1.2	Saída	15
3.1.3	E/S Exemplo	15
3.1.4	Entrada e Saída Formatada	16

3.1.5	Entrada e Saída de Dados em Arquivos	16
3.2	Seleção e condicional	17
3.3	Repetição	18
3.3.1	while/do-while	18
3.3.2	for/foreach	18
3.4	Funções	19
3.5	Módulos e Subprogramas	19
3.5.1	Objetos	19
3.5.2	Classes	19
3.5.3	Metodos	20
3.5.4	Funções	20
3.5.5	Subprogramas	21
4	Aplicações da Linguagem Scala	23
4.1	QuickSort	23
4.2	Fatorial em Scala	24
4.3	Verificar Numero primo Scala	26
4.4	Fibonacci em Scala	28
4.5	Media de uma Lista	28
5	Ferramentas existentes e utilizadas	31
5.1	Editores para Scala	31
5.2	Compiladores	32
5.3	Ambientes de Programação IDE para Scala	33
6	Considerações Finais	35
6.1	Conclusão	35
6.1.1	Problemas enfrentados	35
6.1.2	Resumo do trabalho desenvolvido	35
6.1.3	Aspectos não considerados e sugestões para trabalhos futuros	36
	Bibliografia	37

1. Introdução

Scala é uma linguagem de programação moderna e **multi-paradigma** desenvolvida para expressar padrões de programação comuns em uma forma concisa, elegante e com tipagem segura. Integra facilmente características de linguagens orientadas a objetos e funcional.

1.1 Aspectos históricos da linguagem Scala

Scala é uma linguagem de programação de uso geral. Foi criada e desenvolvida por Martin Odersky. Martin começou a trabalhar no Scala em 2001 na École Polytechnique Fédérale de Lausanne (EPFL). Foi lançado oficialmente em 20 de janeiro de 2004.

Scala não é uma extensão do Java, mas é totalmente interoperável com ele. Durante a compilação, o arquivo Scala é traduzido para bytecode Java e executado em JVM (Java Virtual machine).

"Eu ainda queria combinar programação funcional e orientada a objetos, mas sem as restrições impostas pelo Java. Eu havia descoberto sobre o cálculo de junção e acreditava que essa seria uma excelente base para basear tal unificação. O resultado foi Funnel, uma linguagem de programação para redes funcionais. Este era um design lindamente simples, com poucos recursos de linguagem primitiva.(...)

Contudo, descobriu-se que a linguagem não era muito agradável de usar na prática. O minimalismo é ótimo para designers de linguagem, mas não para usuários. Os usuários não especialistas não sabem como fazer as codificações necessárias, e os usuários experientes ficam entediados de ter que fazê-las repetidas vezes. Além disso, tornou-se rapidamente evidente que qualquer nova linguagem só terá chance de ser aceita se vier com um grande conjunto de bibliotecas padrão." [BV09]

Scala foi projetado para ser **orientado a objetos e funcional**. É uma linguagem pura orientada a objetos no sentido de que todo valor é um objeto e uma linguagem funcional no sentido de que toda função é um valor. O nome scala é derivado da palavra escalável, o que significa que pode crescer com a demanda dos usuários.

1.2 Áreas de Aplicação da Linguagem

Neste capítulo serão apresentadas as principais áreas de aplicação da linguagem Scala.

1.2.1 Desenvolvimento Web



Figure 1.1: Web Dev, Autor: PNGWING

O desenvolvimento web com Scala envolve a criação de aplicativos web dinâmicos e escaláveis. Frameworks como Play Framework e Lift são amplamente utilizados para construir aplicativos web modernos. Scala oferece uma sintaxe concisa e expressiva, facilitando o desenvolvimento rápido e seguro de aplicativos web complexos.

1.2.2 Processamento de Big Data



Figure 1.2: Big Data, Autor: PNGWING

Big data refere-se ao processo de análise, processamento e interpretação de grandes conjuntos de dados. Scala é uma escolha popular para o processamento de big data, especialmente em projetos que envolvem tecnologias como Apache Spark, Apache Flink e Akka. Sua sintaxe concisa e expressiva é adequada para lidar com pipelines de processamento de dados complexos e distribuídos.

1.2.3 Machine Learning e Inteligência Artificial

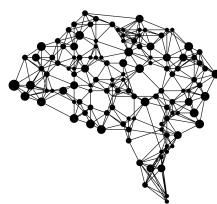


Figure 1.3: Rede Neural, Autor: PNGWING

Machine learning e inteligência artificial envolvem o desenvolvimento de algoritmos e modelos que podem aprender e tomar decisões com base em dados. Scala é frequentemente utilizada em projetos de machine learning e inteligência artificial devido à sua capacidade de lidar com algoritmos complexos e manipulação eficiente de grandes conjuntos de dados. Bibliotecas como Apache Mahout e Breeze oferecem suporte para desenvolvimento de modelos de machine learning em Scala.

1.2.4 Desenvolvimento de Aplicações Desktop e Mobile

Scala também pode ser utilizada no desenvolvimento de aplicativos desktop e mobile. Frameworks como JavaFX e ScalaFX permitem aos desenvolvedores criar interfaces gráficas de usuário utilizando a linguagem Scala. Sua integração com o ambiente de desenvolvimento Java oferece acesso a uma ampla gama de bibliotecas e ferramentas para criar aplicativos interativos e visualmente atraentes.

1.2.5 Finanças e Serviços Financeiros



Figure 1.4: Finance, Autor: PNGWING

Na área financeira, Scala é frequentemente utilizada para análise de dados, modelagem financeira e desenvolvimento de sistemas de negociação. Sua capacidade de lidar com cálculos complexos e manipulação eficiente de dados financeiros torna-a uma escolha popular entre empresas que trabalham com análise de dados financeiros e gerenciamento de riscos.

1.2.6 Educação e Pesquisa

Scala é uma escolha popular em ambientes acadêmicos e de pesquisa, onde é utilizada para ensinar conceitos avançados de programação e realizar pesquisas em áreas como linguagens de programação, algoritmos e sistemas distribuídos. Sua combinação única de programação orientada a objetos e funcional torna-a uma ferramenta poderosa para explorar conceitos de computação moderna e desenvolver soluções inovadoras para problemas computacionais complexos.

2. Conceitos básicos da Linguagem Scala

2.1 Variáveis e constantes

Scala é uma linguagem com tipagem estática, diferente de Python, por exemplo, em que sua tipagem é dinâmica. Isso significa que uma **variável em Scala não pode ter seu tipo mudado**. Geralmente, em linguagens dinâmicas, o valor de uma variável é processado durante o tempo de execução, enquanto nas linguagens estáticas, o valor é processado no tempo de parse. Em questão de performance, otimizar um código em linguagem dinâmica é muito mais desafiador do que em linguagem estática [Wam21].

Existem duas declarações de variáveis em Scala: ‘val’, que é constante, e ‘var’, que pode ser mudada:

"val" Quando a variável é imutável (Read-Only)

```
1 Scala> val Numero: Int = 1
```

"var" Quando a variável é mutável (Read and Write)

```
1 Scala> var Numero: Int = 1
2 resultado: Numero = 1
3 Scala> Numero = 2
4 resultado: Numero = 2
```

Scala recomenda que você use as variáveis do tipo **val** sempre que possível, pois isso promove um melhor design orientado a objeto e é consistente com o conceito de programação funcional "pura" [Wam21].

2.2 Tipos de Dados

Os tipos de dados são fundamentais em qualquer linguagem de programação, pois definem a natureza dos dados que podem ser manipulados. Em Scala, os tipos de dados são amplamente categorizados em tipos de dados primitivos e compostos.

Uma visão geral dos tipos de dados:

```

1   object TiposDeDados {
2     def main(args: Array[String]): Unit = {
3       // Tipos de dados primitivos
4       val inteiro: Int = 42
5       val grandeInteiro: Long = 123456789L
6       val pontoFlutuante: Float = 3.14f
7       val grandePontoFlutuante: Double = 2.71828
8       val caractere: Char = 'a'
9       val booleano: Boolean = true
10      val string: String = "Hello, Scala!"
11
12      // Tipos de dados compostos
13      val array: Array[Int] = Array(1, 2, 3, 4)
14      val lista: List[Int] = List(1, 2, 3, 4)
15      val conjunto: Set[Int] = Set(1, 2, 3, 4)
16      val mapa: Map[String, String] =
17        Map("key1" -> "value1", "key2" -> "value2")
18      val tupla: (Int, String, Boolean) = (1, "Scala", true)
19
20      // Imprimindo os valores
21      println(s"Int: $inteiro")
22      println(s"Long: $grandeInteiro")
23      println(s"Float: $pontoFlutuante")
24      println(s"Double: $grandePontoFlutuante")
25      println(s"Array: ${array.mkString(", ")}")
26      ...
27    }

```

2.2.1 Tipo de Dados Primitivos

Os tipos de dados primitivos em Scala são os blocos de construção **fundamentais para manipulação de dados básicos**. Esses tipos são usados para armazenar valores simples, como números e caracteres, e são essenciais para a criação de qualquer programa.[Wam21]

String

Uma String em Scala é uma **sequência de caracteres imutável**, o que significa que seu conteúdo não pode ser alterado depois de criada. Strings são usadas para representar texto e são amplamente utilizadas em programação para manipulação de dados textuais. Em Scala, a classe String é baseada na classe String do Java, o que significa que todas as funcionalidades da classe Java String estão disponíveis em Scala.

```

1  val str: String = "Hello, Scala!"
2  println(str)

```

Int

O tipo Int em Scala representa **números inteiros** de 32 bits. Ele é usado para armazenar valores inteiros, que podem variar de -2 elevado a 31 a 2 elevado a 31-1. Em Scala, Int é um objeto, o que significa que pode ser usado com métodos e operações como qualquer outro objeto.

Exemplo:

```
1 val num: Int = 10
```

Float

O tipo Float é usado para representar números de **ponto flutuante de precisão simples** de 32 bits, conforme definido pelo padrão IEEE 754. Isso permite armazenar valores de ponto flutuante com uma precisão limitada. Em Scala, Float é um objeto, permitindo operações aritméticas e acesso a métodos. De forma mais simples podemos chama-los de numeros do conjunto real(3.14 , 0.2, 32.911) lembrando da limitação de memoria já que o computador não trabalha com infinito.

```
1 val numFloat: Float = 3.14
```

Boolean

O tipo Boolean em Scala representa valores verdadeiro (true) ou falso (false). É usado para **operações lógicas e condicionais**. Assim como os outros tipos, Boolean é um objeto em Scala, permitindo operações lógicas e acesso a métodos.

```
1 val isTrue: Boolean = true
```

Char

O tipo Char é usado para representar **um único caractere** Unicode de 16 bits. Isso permite armazenar qualquer caractere Unicode dentro de um único valor. Em Scala, Char é um objeto, permitindo operações com caracteres e acesso a métodos.

Exemplo:

```
1 val letra: Char = 'a'
```

Byte

O tipo Byte é usado para representar **valores inteiros de 8 bits**. Ele pode armazenar valores de -128 a 127. Em Scala, Byte é um objeto, permitindo operações aritméticas e acesso a métodos.

Exemplo:

```
1 val numByte: Byte = 127
```

Short

O tipo Short é usado para representar **valores inteiros de 16 bits**. Ele pode armazenar valores de -32768 a 32767. Em Scala, Short é um objeto, permitindo operações aritméticas e acesso a métodos.

Exemplo:

```
1 val numShort: Short = 32767
```

Long

O tipo Long é usado para representar **valores inteiros de 64 bits**. Ele pode armazenar valores de -2⁶³ a 2⁶³-1. Em Scala, Long é um objeto, permitindo operações aritméticas e acesso a métodos.

Exemplo:

```
1 val numLong: Long = 9223372036854775807L
```

Double

O tipo Double é usado para representar números de **ponto flutuante de precisão dupla** de 64 bits, conforme definido pelo padrão IEEE 754. Isso permite armazenar valores de ponto flutuante com uma precisão maior do que o Float. Em Scala, Double é um objeto, permitindo operações aritméticas e acesso a métodos.

Exemplo:

```
1 val numDouble: Double = 3.141592653589793
```

Unit

O tipo Unit em Scala é usado para representar um valor que não **carrega informação significativa**. É o tipo de retorno padrão para funções que não retornam um valor. Em Scala, Unit é um objeto, permitindo operações e acesso a métodos.

Exemplo:

```
1 def printMessage(): Unit = println("Hello, World!")
```

2.2.2 Tipos de Dados Compostos

Os tipos de dados compostos em Scala permitem agrupar múltiplos valores em uma única entidade, facilitando a manipulação de coleções de dados mais complexas.

Array

Um array (vetor) é uma **coleção de elementos do mesmo tipo**, com tamanho fixo. Permite acesso rápido aos elementos usando índices.

```
1 val array: Array[Int] = Array(1, 2, 3, 4)
2 println(s"Array: ${array.mkString(", ")}")
```

Lista

Uma lista é uma **coleção imutável de elementos**, onde cada elemento aponta para o próximo.

```
1 val list: List[Int] = List(1, 2, 3, 4)
2 println(s"List: $list")
```

Set

Um conjunto (set) é uma **coleção imutável de elementos únicos**. Útil para operações matemáticas de conjunto, como união e interseção.

```
1 object ExemploUniaoSet {
2     def main(args: Array[String]): Unit = {
3         val set1: Set[Int] = Set(1, 2, 3)
4         val set2: Set[Int] = Set(3, 4, 5)
5
6         // Uni o usando o método union
7         val uniao1: Set[Int] = set1.union(set2)
8         println(s"Uni o usando union: $uniao1")
9
10        // Uni o usando o operador ++
11        val uniao2: Set[Int] = set1 ++ set2
12        println(s"Uni o usando ++: $uniao2")
```

```
13   }
14 }
```

Map

Um mapa (map) é uma coleção de pares chave-valor, onde cada chave é única. Permite acesso rápido aos valores associados a uma chave.

```
1 val map: Map[String, Int] = Map("a" -> 1, "b" -> 2)
2 println(s"Map: $map")
```

Tuple

Uma tupla (tuple) é uma **coleção de elementos de diferentes tipos** agrupados juntos. Útil para retornar múltiplos valores de uma função.

```
1 val tuple: (Int, String, Boolean) = (1, "Scala", true)
2 println(s"Tuple: $tuple")
```

2.3 Operadores em Scala

Os operadores em Scala são usados para **realizar várias operações** em variáveis e valores. Scala suporta muitos operadores comuns encontrados em outras linguagens de programação, como aritméticos, relacionais e lógicos.

2.3.1 Operadores Aritméticos

Os operadores aritméticos são utilizados para realizar operações **matemáticas** básicas.

- Adição (+): Soma dois valores.
- Subtração (-): Subtrai um valor de outro.
- Multiplicação (*): Multiplica dois valores.
- Divisão (/): Divide um valor por outro.
- Módulo (%): Retorna o resto da divisão entre dois valores.

```
1 val soma = 10 + 5 // soma    15
2 val subtracao = 20 - 7 // subtracao  13
3 val multiplicacao = 8 * 4 // multiplicacao  32
4 val divisao = 100 / 5 // divisao    20
5 val modulo = 15 % 4 // modulo     3
```

2.3.2 Operadores de Comparação

Os operadores de comparação são utilizados para comparar valores e produzir resultados booleanos (verdadeiro ou falso).

- Igualdade (==): Verifica se dois valores são iguais.
- Diferença (!=): Verifica se dois valores são diferentes.
- Maior que (>): Verifica se um valor é maior que outro.
- Menor que (<): Verifica se um valor é menor que outro.
- Maior ou igual (>=): Verifica se um valor é maior ou igual a outro.
- Menor ou igual (<=): Verifica se um valor é menor ou igual a outro.

```
1 val igualdade = 10 == 10 // igualdade  true
2 val diferenca = 20 != 15 // diferenca  true
```

```

3 val maiorQue = 30 > 25 // maiorQue      true
4 val menorQue = 40 < 35 // menorQue      false
5 val maiorOuIgual = 50 >= 50 // maiorOuIgual    true
6 val menorOuIgual = 60 <= 55 // menorOuIgual   false

```

2.3.3 Operadores Lógicos

Os operadores lógicos são utilizados para combinar expressões booleanas.

- E lógico (&&): Retorna verdadeiro se ambas as expressões forem verdadeiras.

A	B	$A \wedge B$
Verdadeiro (V)	Verdadeiro (V)	Verdadeiro (V)
Verdadeiro (V)	Falso (F)	Falso (F)
Falso (F)	Verdadeiro (V)	Falso (F)
Falso (F)	Falso (F)	Falso (F)

- Ou lógico (||): Retorna verdadeiro se pelo menos uma das expressões for verdadeira.

A	B	$A \vee B$
Verdadeiro (V)	Verdadeiro (V)	Verdadeiro (V)
Verdadeiro (V)	Falso (F)	Verdadeiro (V)
Falso (F)	Verdadeiro (V)	Verdadeiro (V)
Falso (F)	Falso (F)	Falso (F)

- Não lógico (!): Inverte o valor de uma expressão.

A	$\neg A$
Verdadeiro (V)	Falso (F)
Falso (F)	Verdadeiro (V)

```

1 val expressao1 = true
2 val expressao2 = false
3 val eLogico = expressao1 && expressao2 // eLogico      false
4 val ouLogico = expressao1 || expressao2 // ouLogico    true
5 val naoLogico = !expressao1 // naoLogico   false

```

3. Programação em Scala

3.1 Entradas e Saídas

As entradas e saídas de dados em Scala são fundamentais para a interação entre programas Scala e seus usuários ou sistemas externos. A principal e mais básica forma de entrada de dados e exibição de respostas é o console.

3.1.1 Entrada

```
1 val input = scala.io.StdIn.readLine("Digite alguma coisa:")
```

Assim que o programa for executado e chegar nessa linha, ele irá aguardar o usuário **inserir a informação através do console**. Após isso, o usuário aperta Enter e confirma o que foi digitado, que é enviado para um espaço de memória.

3.1.2 Saída

```
1 println("Hello, World!")
```

Através dessa linha, o que está entre parênteses é impresso (mostrado) na janela do console.

3.1.3 E/S Exemplo

```
1 println("Digite o primeiro numero:")
2 val numero1 = readInt()
3
4 // Solicitar o segundo numero ao usuario
5 println("Digite o segundo numero:")
6 val numero2 = readInt()
7
8 // Calcular a soma dos dois numeros
9 val soma = numero1 + numero2
```

```

10 // Solicitar ao usuario a resposta esperada
11 println(s"A soma de $numero1 e $numero2      $soma.
12 Digue a resposta correta:")
13
14 // Ler a resposta do usuario
15 val respostaUsuario = readInt()
16

```

Nesse trecho de código, podemos ver os dois comandos sendo usados de forma simples. O computador imprime as mensagens para o usuário digitar dois números, lê e armazena esses números e, em seguida, o usuário digita a soma deles. Apesar de o computador processar a soma desses números na linha ‘val soma = numero1 + numero2‘, ele ainda não mostrou ao usuário se a soma que ele colocou está correta.

3.1.4 Entrada e Saída Formatada

A formatação de strings é essencial para apresentar informações de maneira clara e organizada. Em Scala, existem dois principais métodos: ‘format()‘ e ‘formatted()‘.

O método ‘format()‘ permite formatar strings, aceitando parâmetros que substituem placeholders na string. Os placeholders são indicados por % seguido por um especificador de formato, como %d para inteiros e %f para pontos flutuantes ou doubles.

Exemplo:

```

1 // Criando uma string formatada
2 val x = "There are %d books and the cost of each book is %f"
3
4 // Atribuindo valores
5 val y = 15
6 val z = 345.25
7
8 // Aplicando o m todo format
9 val r = x.format(y, z)
10
11 // Exibindo a string formatada
12 println(r)

```

3.1.5 Entrada e Saída de Dados em Arquivos

Para ler um arquivo de texto, você pode usar o método ‘Source‘ da classe ‘Source‘ do pacote ‘scala.io.Source‘:

```

1 import scala.io.Source
2 val fileContent = Source.fromFile("/path/to/file.txt").
3     mkString
4 println(fileContent)

```

Através desse código, todo o conteúdo dentro do arquivo "file.txt" é mostrado no console de forma que é possível ler seu conteúdo.

Para escrever em um arquivo, você pode usar o método ‘write‘ da classe ‘PrintWriter‘:

```

1 import java.io.PrintWriter
2
3 val writer = new PrintWriter(new File("/path/to/file.txt"))

```

```

4 writer.write("Hello , Scala!")
5 writer.close()

```

Nesse código, o arquivo "file.txt" é aberto pelo programa, e nele é escrito "Hello, Scala!" depois o arquivo é fechado.

3.2 Seleção e condicional

Em Scala, você pode usar a estrutura condicional **if(em portugues:se)** para executar diferentes blocos de código com base em uma condição booleana. Além disso, Scala oferece uma variedade de formas de expressar condições complexas e múltiplas usando if, else if e else, bem como o match, que é semelhante ao switch em outras linguagens de programação. Ela pega emprestado a maioria dos operadores do Java[Wam21].

O **else**(portugues: senão) serve para quando a condição do if não é satisfeita, desse modo ele é executado no lugar, veremos os exemplos.

if, else if, else A estrutura básica do if em Scala é a seguinte:

```

1   if (condicao) {
2     // bloco de código executado se a condição for verdadeira
3   } else if (outraCondicao) {
4     // bloco de código executado se a outra condição for verdadeira
5   } else {
6     // bloco de código executado se
7     //nenhuma das condições anteriores for verdadeira
8   }
9

```

exemplo de condição:

```

1   val x = 10
2
3   if (x > 0) {
4     println("x eh positivo")
5   } else if (x < 0) {
6     println("x eh negativo")
7   } else {
8     println("x eh zero")
9   }

```

match

O match em Scala é semelhante ao switch em outras linguagens de programação, mas é muito mais poderoso e flexível. Ele pode ser usado para comparar um valor com uma série de padrões e executar o bloco de código correspondente ao padrão que corresponde ao valor.

```

1   val diaDaSemana = "segunda-feira"
2
3   val mensagem = diaDaSemana match {
4     case "segunda-feira" | "terça-feira"
5     | "quarta-feira" | "quinta-feira" | "sexta-feira" =>
6       "Dia de trabalho"
7     case "sábado" | "domingo" =>

```

```

8     "Fim de semana"
9     case _ =>
10    "Dia invalido"
11  }
12  println(mensagem)

```

No exemplo acima, o match compara diaDaSemana com diferentes padrões e executa o bloco de código correspondente ao padrão que coincide. O `_` é um padrão curinga que corresponde a qualquer valor.

3.3 Repetição

Também conhecido como **loop**, é referido como a capacidade de executar um bloco de código repetidamente com base em certas condições. Esse conceito é fundamental para **automatizar** tarefas que precisam ser executadas várias vezes sem a necessidade de repetir manualmente o mesmo código.

3.3.1 while/do-while

O while é uma estrutura de repetição que executa um bloco de código enquanto uma condição específica for verdadeira.

```

1 var i = 0
2 while (i < 5) {
3   println(s"Valor de i: $i")
4   i += 1
5 }

```

O do-while é semelhante ao while, mas garante que o bloco de código seja **executado pelo menos uma vez**, mesmo que a condição seja falsa na primeira vez. `var j = 0` do `println(s"Valor de j: $j")` `j += 1` `while (j < 5)`

3.3.2 for/foreach

O **for** em Scala pode ser usado para percorrer uma sequência de valores, como uma faixa numérica ou uma coleção.

```

1 for (i <- 0 until 5) {
2   println(s"Valor de i: $i")
3 }

```

Este loop também imprimirá os valores de `i` de 0 a 4. A expressão `0 until 5` cria uma faixa de números de 0 a 4, e o loop itera sobre esses valores.

O **foreach** é usado para percorrer elementos em uma coleção, como listas, arrays ou sequências.

```

1 val lista = List("a", "b", "c", "d", "e")
2 lista.foreach { elemento =>
3   println(s"Elemento: $elemento")
4 }

```

Este loop imprimirá cada elemento da lista, de "a" a "e". O método `foreach` é chamado em uma coleção e uma função é passada como argumento. Essa função é então aplicada a cada elemento da coleção.

3.4 Funções

Funções são estruturas de código que tem o objetivo de tornar o código mais encapsulado, elas são amplamente utilizadas em trechos que podem se repetir muitas vezes em um código, ou em partes diferentes do programa. exemplo:

```
1  def somar(a: Int, b: Int): Int = {
2      a + b
3  }
```

Neste exemplo, a função somar recebe dois parâmetros inteiros a e b e retorna a soma deles como um inteiro.

Chamando Funções Depois de definir uma função, você pode chamá-la em qualquer lugar do seu código, fornecendo os valores necessários para os parâmetros.

scala Copiar código

```
1 val resultado = somar(3, 5) // resultado     8
```

Neste exemplo, estamos chamando a função somar com os valores 3 e 5 e atribuindo o resultado a uma variável chamada resultado.

3.5 Módulos e Subprogramas

3.5.1 Objetos

Os objetos em Scala são instâncias únicas de suas próprias definições, similares aos singletons em outras linguagens de programação. Eles são usados para agrupar funções e valores que pertencem logicamente juntos.

```
1  object Utilidades {
2      def saudacao(nome: String): String = s"Olá, $nome!"
3
4      def soma(a: Int, b: Int): Int = a + b
5  }
6
7  object TesteUtilidades {
8      def main(args: Array[String]): Unit = {
9          println(Utilidades.saudacao("Mundo"))
10         println(s"A soma de 2 e 3 é ${Utilidades.soma(2, 3)}")
11     }
12 }
```

este exemplo, Utilidades é um objeto que contém duas funções: saudacao e soma. O objeto TesteUtilidades demonstra como chamar essas funções.

3.5.2 Classes

As classes em Scala são usadas para definir tipos de dados personalizados e podem conter métodos e variáveis. As instâncias dessas classes são criadas usando a palavra-chave new.

```
1  class Pessoa(val nome: String, val idade: Int) {
2      def saudacao(): String = s"Olá, meu nome
3          $nome e eu tenho $idade anos."
4  }
5
```

```

6 object TestePessoa {
7     def main(args: Array[String]): Unit = {
8         val pessoa = new Pessoa("Alice", 30)
9         println(pessoa.saudacao())
10    }
11 }

```

3.5.3 Métodos

Os métodos são definidos dentro de classes ou objetos e são semelhantes às funções, mas podem acessar os campos da classe ou do objeto.

```

1   class Calculadora {
2       def soma(a: Int, b: Int): Int = a + b
3       def subtracao(a: Int, b: Int): Int = a - b
4   }
5
6 object TesteCalculadora {
7     def main(args: Array[String]): Unit = {
8         val calculadora = new Calculadora
9         println(s"2 + 3 = ${calculadora.soma(2, 3)}")
10        println(s"5 - 2 = ${calculadora.subtracao(5,
11            2)}")
12    }
13 }

```

3.5.4 Funções

As funções em Scala podem ser definidas de forma independente ou como parte de objetos e classes. Funções anônimas (ou literais de função) são funções sem nome que podem ser atribuídas a variáveis ou passadas como argumentos.

```

1   object Funcoes {
2       val soma: (Int, Int) => Int = (a, b) => a + b
3       val saudacao: String => String = nome => s"Olá, $nome!"
4
5       def aplicarFuncao(f: Int => Int, x: Int): Int = f(x)
6   }
7
8 object TesteFuncoes {
9     def main(args: Array[String]): Unit = {
10         println(Funcoes.saudacao("Scala"))
11         println(s"A soma de 3 e 4 é ${Funcoes.soma(3, 4)}")
12         val dobro = (x: Int) => x * 2
13         println(s"O dobro de 5 é ${Funcoes.aplicarFuncao(
14             dobro, 5)}")
15     }
16 }

```

Neste exemplo, Funcoes é um objeto que define duas funções como valores (soma e saudacao) e um método aplicarFuncao que aceita uma função como argumento.

3.5.5 Subprogramas

A modularização em Scala pode ser alcançada dividindo o código em vários objetos, classes e pacotes. Um pacote é um mecanismo para agrupar código relacionado, semelhante a um namespace em outras linguagens.

```
1 package utilidades
2
3 object Conversores {
4     def celsiusParaFahrenheit(celsius: Double): Double =
5         celsius * 9/5 + 32
6     def fahrenheitParaCelsius(fahrenheit: Double): Double =
7         (fahrenheit - 32) * 5/9
8 }
9
10 object TesteConversores {
11     def main(args: Array[String]): Unit = {
12         println(s"25 C = ${utilidades.Conversores.
13             celsiusParaFahrenheit(25)} F ")
14         println(s"77 F = ${utilidades.Conversores.
15             fahrenheitParaCelsius(77)} C ")
16     }
17 }
```

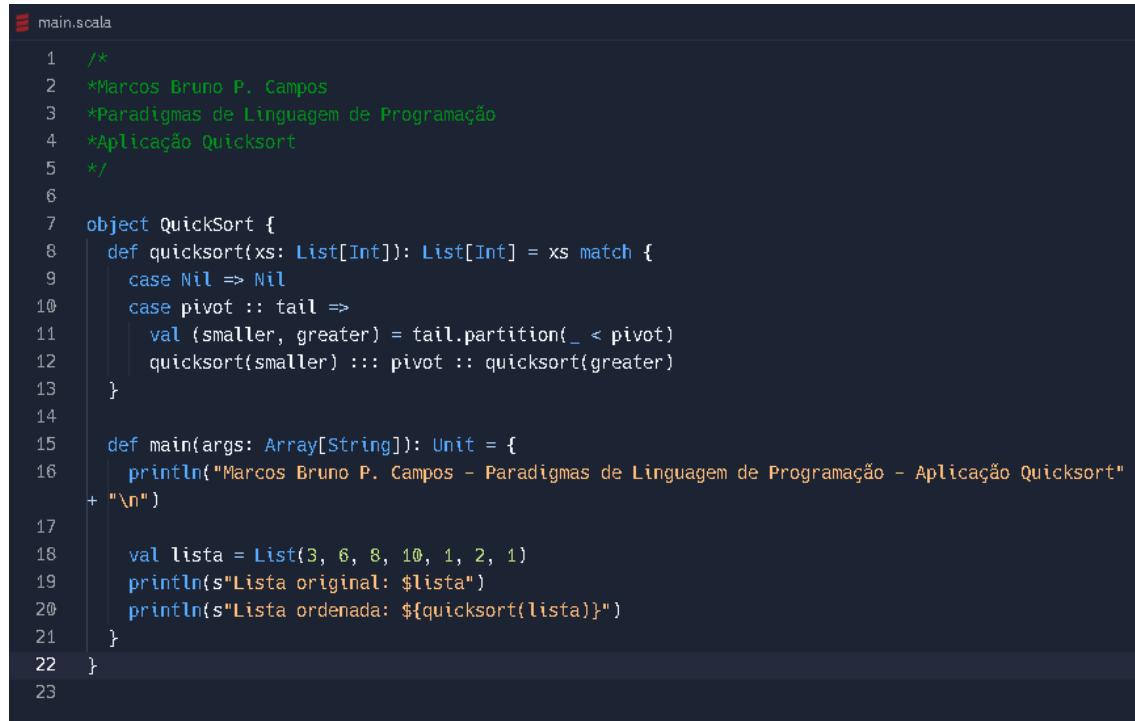
Neste exemplo, criamos um pacote utilidades que contém o objeto Conversores com métodos de conversão de temperatura.

4. Aplicações da Linguagem Scala

4.1 QuickSort

O QuickSort é um dos algoritmos de ordenação mais eficientes, amplamente utilizado devido à sua complexidade média de $O(n \log n)$. Ele funciona selecionando um 'pivô' e particionando a lista de forma que todos os elementos menores que o pivô fiquem à sua esquerda e todos os elementos maiores, à sua direita. Este processo é então recursivamente aplicado às sublistas esquerda e direita.

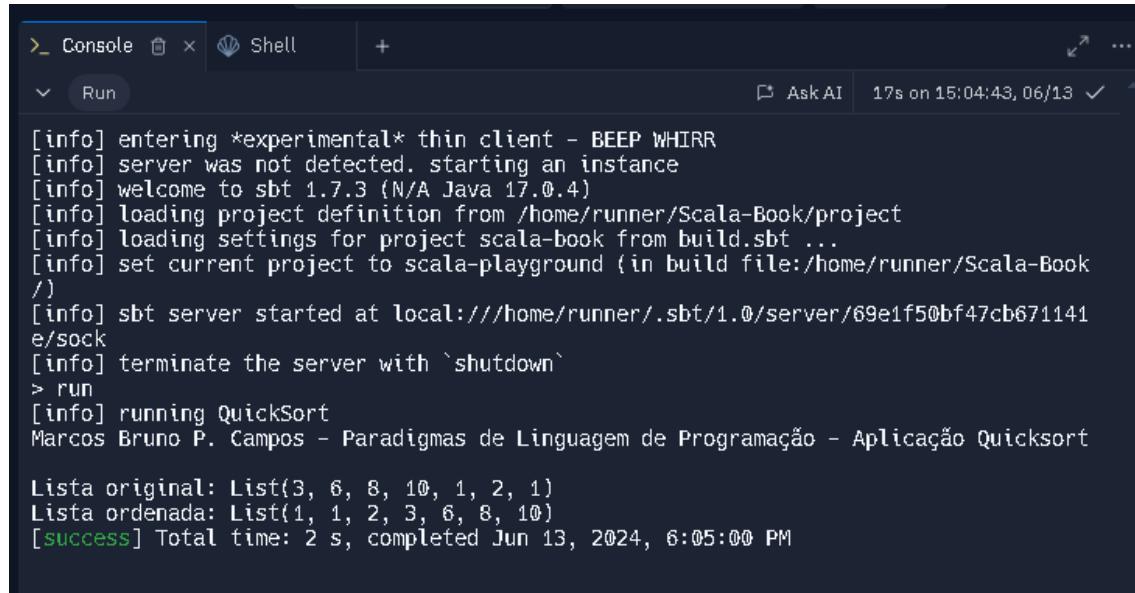
```
1  object QuickSort {
2    def quicksort(xs: List[Int]): List[Int] = xs match {
3      case Nil => Nil
4      case pivot :: tail =>
5        val (smaller, greater) = tail.partition(_ < pivot)
6        quicksort(smaller) ::: pivot :: quicksort(greater)
7    }
8
9    def main(args: Array[String]): Unit = {
10      val lista = List(3, 6, 8, 10, 1, 2, 1)
11      println(s"Lista original: $lista")
12      println(s"Lista ordenada: ${quicksort(lista)}")
13    }
14 }
```



```

1  /*
2  *Marcos Bruno P. Campos
3  *Paradigmas de Linguagem de Programação
4  *Aplicação Quicksort
5  */
6
7  object QuickSort {
8      def quicksort(xs: List[Int]): List[Int] = xs match {
9          case Nil => Nil
10         case pivot :: tail =>
11             val (smaller, greater) = tail.partition(_ < pivot)
12             quicksort(smaller) ::: pivot :: quicksort(greater)
13     }
14
15     def main(args: Array[String]): Unit = {
16         println("Marcos Bruno P. Campos - Paradigmas de Linguagem de Programação - Aplicação Quicksort"
17 + "\n")
18
19         val lista = List(3, 6, 8, 10, 1, 2, 1)
20         println(s"Lista original: $lista")
21         println(s"Lista ordenada: ${quicksort(lista)}")
22     }
23 }
```

Figure 4.1: Quicksort no replit



```

>_ Console  x  Shell  +  ↗ Ask AI  17s on 15:04:43, 06/13 ✓ ...
Run
[info] entering *experimental* thin client - BEEP WHIRR
[info] server was not detected. starting an instance
[info] welcome to sbt 1.7.3 (N/A Java 17.0.4)
[info] loading project definition from /home/runner/Scala-Book/project
[info] loading settings for project scala-book from build.sbt ...
[info] set current project to scala-playground (in build file:/home/runner/Scala-Book/)
[info] sbt server started at local:///home/runner/.sbt/1.0/server/69e1f50bf47cb671141
e.sock
[info] terminate the server with `shutdown`
> run
[info] running QuickSort
Marcos Bruno P. Campos - Paradigmas de Linguagem de Programação - Aplicação Quicksort

Lista original: List(3, 6, 8, 10, 1, 2, 1)
Lista ordenada: List(1, 1, 2, 3, 6, 8, 10)
[success] Total time: 2 s, completed Jun 13, 2024, 6:05:00 PM
```

Figure 4.2: Terminal no replit

4.2 Fatorial em Scala

O fatorial de um número é o produto de todos os números inteiros positivos até aquele número. É frequentemente usado em matemática e ciência da computação. Em Scala, podemos calcular o fatorial de um número usando recursão, onde a função chama a si mesma com um valor decrementado até atingir a base do caso ($n == 0$).

```
1  object Fatorial {  
2    def fatorial(n: Int): Int = {  
3      if (n == 0) 1  
4      else n * fatorial(n - 1)  
5    }  
6  
7    def main(args: Array[String]): Unit = {  
8      val numero = 5  
9      println(s"O factorial de $numero é ${fatorial(numero)}")  
10     }  
11 }
```

The screenshot shows a code editor window with a dark theme. The file is named 'main.scala'. The code is as follows:

```
1  /*  
2   *Marcos Bruno P. Campos  
3   *Paradigmas de Linguagem de Programação  
4   *Aplicação Fatorial  
5   */  
6  
7  object Fatorial {  
8    def fatorial(n: Int): Int = {  
9      if (n == 0) 1  
10     else n * fatorial(n - 1)  
11   }  
12  
13  def main(args: Array[String]): Unit = {  
14    val numero = 5  
15    println("Marcos Bruno P. Campos ")  
16    println(s"O factorial de $numero é ${fatorial(numero)}")  
17  }  
18}  
19
```

Figure 4.3: Fatorial no replit

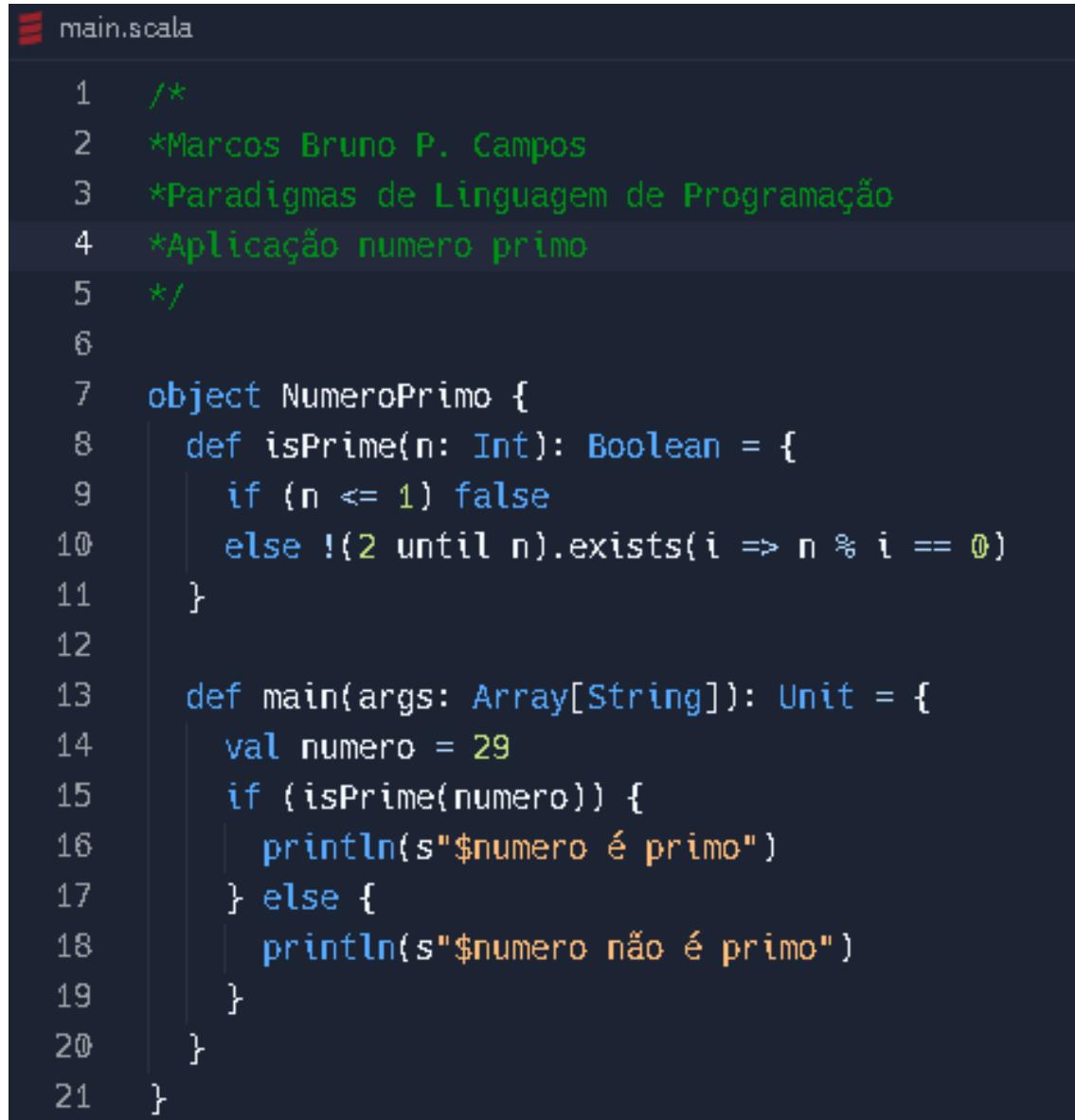
```
[info] entering *experimental* thin client - BEEP WHIRR
[info] server was not detected. starting an instance
[info] welcome to sbt 1.7.3 (N/A Java 17.0.4)
[info] loading project definition from /home/runner/Scala-Book/project
[info] loading settings for project scala-book from build.sbt ...
[info] set current project to scala-playground (in build file:/home/runner/Scala-Book/)
[info] sbt server started at local://home/runner/.sbt/1.0/server/69e1f50bf47cb671141e/sock
[info] st[info] terminate the server with `shutdown`
> run
[info] compiling 1 Scala source to /home/runner/Scala-Book/target/scala-2.12/classes ...
[info] running Fatorial
Marcos Bruno P. Campos
0 fatorial de 5 é 120
[success] Total time: 9 s, completed Jun 13, 2024, 6:15:23 PM
```

Figure 4.4: Fatorial no terminal

4.3 Verificar Número primo Scala

Um número primo é um número inteiro maior que 1 que não tem divisores positivos além de 1 e ele mesmo. Verificar se um número é primo envolve testar se ele é divisível por qualquer número entre 2 e sua raiz quadrada. Em Scala, podemos implementar essa verificação usando uma função que testa divisibilidade.

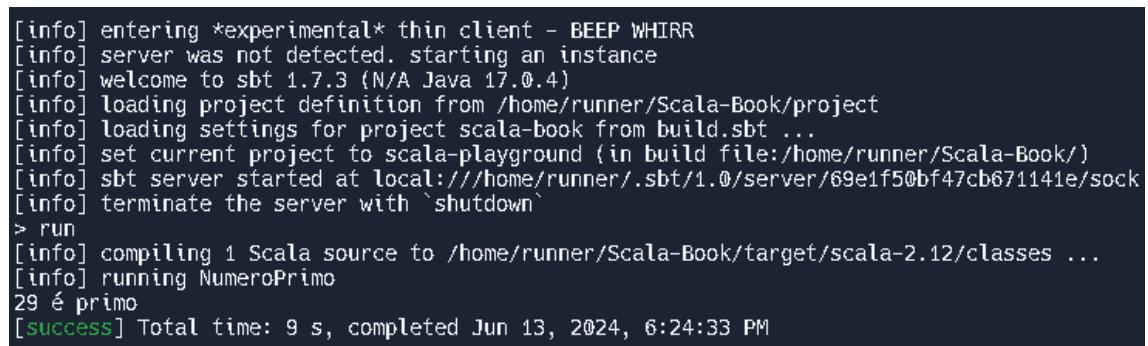
```
1   object NumeroPrimo {
2     def isPrime(n: Int): Boolean = {
3       if (n <= 1) false
4       else !(2 until n).exists(i => n % i == 0)
5     }
6
7     def main(args: Array[String]): Unit = {
8       val numero = 29
9       if (isPrime(numero)) {
10         println(s"$numero      primo")
11       } else {
12         println(s"$numero      n o      primo")
13       }
14     }
15 }
```



```
main.scala

1  /*
2   *Marcos Bruno P. Campos
3   *Paradigmas de Linguagem de Programação
4   *Aplicação numero primo
5  */
6
7  object NumeroPrimo {
8      def isPrime(n: Int): Boolean = {
9          if (n <= 1) false
10         else !(2 until n).exists(i => n % i == 0)
11     }
12
13    def main(args: Array[String]): Unit = {
14        val numero = 29
15        if (isPrime(numero)) {
16            println(s"$numero é primo")
17        } else {
18            println(s"$numero não é primo")
19        }
20    }
21 }
```

Figure 4.5: código numero primo



```
[info] entering *experimental* thin client - BEEP WHIRR
[info] server was not detected. starting an instance
[info] welcome to sbt 1.7.3 (N/A Java 17.0.4)
[info] loading project definition from /home/runner/Scala-Book/project
[info] loading settings for project scala-book from build.sbt ...
[info] set current project to scala-playground (in build file:/home/runner/Scala-Book/)
[info] sbt server started at local:///home/runner/.sbt/1.0/server/69e1f50bf47cb671141e/sock
[info] terminate the server with `shutdown`
> run
[info] compiling 1 Scala source to /home/runner/Scala-Book/target/scala-2.12/classes ...
[info] running NumeroPrimo
29 é primo
[success] Total time: 9 s, completed Jun 13, 2024, 6:24:33 PM
```

Figure 4.6: terminal numero primo

4.4 Fibonacci em Scala

A sequência de Fibonacci é uma série de números onde cada número é a soma dos dois anteriores, começando por 0 e 1. É usada em diversos campos como matemática, computação e biologia. A implementação recursiva em Scala pode ser direta, mas a memoização ou a abordagem iterativa são recomendadas para melhorar a eficiência.

```

1   object Fibonacci {
2     def fibonacci(n: Int): Int = n match {
3       case 0 => 0
4       case 1 => 1
5       case _ => fibonacci(n - 1) + fibonacci(n - 2)
6     }
7
8     def main(args: Array[String]): Unit = {
9       val numero = 10
10      println(s"A sequencia de Fibonacci at $numero : ")
11      for (i <- 0 to numero) {
12        print(s"${fibonacci(i)} ")
13      }
14    }
15 }
```

4.5 Média de uma Lista

Calcular a média de uma lista de números é uma operação comum em estatísticas e análise de dados. A média é a soma de todos os elementos da lista dividida pelo número de elementos. Em Scala, podemos usar as funções integradas da coleção para somar os elementos e calcular a média de maneira eficiente.

```

1   object MediaLista {
2     def media(lista: List[Double]): Double = {
3       lista.sum / lista.length
4     }
5
6     def main(args: Array[String]): Unit = {
7       val numeros = List(10.0, 20.0, 30.0, 40.0, 50.0)
8       println(s"A m dia da lista $numeros ${media(numeros)}")
9     }
10 }
```

```
1  /*
2  *Marcos Bruno P. Campos
3  *Paradigmas de Linguagem de Programação
4  *Aplicação Fibonacci
5  */
6
7  object Fibonacci {
8      def fibonacci(n: Int): Int = n match {
9          case 0 => 0
10         case 1 => 1
11         case _ => fibonacci(n - 1) + fibonacci(n - 2)
12     }
13
14     def main(args: Array[String]): Unit = {
15         val numero = 10
16         println(s"A sequência de Fibonacci até $numero é: ")
17         for (i <- 0 to numero) {
18             print(s"${fibonacci(i)} ")
19         }
20     }
21 }
22
```

Figure 4.7: Fibonacci Código

```
[info] entering *experimental* thin client - BEEP WHIRR
[info] server was not detected. starting an instance
[info] welcome to sbt 1.7.3 (N/A Java 17.0.4)
[info] loading project definition from /home/runner/Scala-Book/project
[info] loading settings for project scala-book from build.sbt ...
[info] set current project to FibonacciApp (in build file:/home/runner/Scala-Book/)
[info] sbt server started at local:///home/runner/.sbt/1.0/server/69e1f50bf47[info] terminate the server with `shutdown`
> run
[info] compiling 1 Scala source to /home/runner/Scala-Book/target/scala-2.13/classes ...
[info] running Fibonacci
A sequência de Fibonacci até 10 é:
0 1 1 2 3 5 8 13 21 34 55 [success] Total time: 9 s, completed Jun 13, 2024, 6:45:20 PM
```

Figure 4.8: Fibonacci Terminal

```

1  /*
2  *Marcos Bruno P. Campos
3  *Paradigmas de Linguagens de Programação
4  * Aplicação Média de listas
5  */
6
7
8  object MediaLista {
9    def media(lista: List[Double]): Double = {
10      lista.sum / lista.length
11    }
12
13   def main(args: Array[String]): Unit = {
14     val numeros = List(10.0, 20.0, 30.0, 40.0, 50.0)
15     println("Marcos Bruno P. Campos")
16     println(s"A média da lista $numeros é ${media(numeros)}")
17   }
18 }
19

```

Figure 4.9: Media Lista Código

```

[info] entering *experimental* thin client - BEEP WHIRR
[info] server was not detected. starting an instance
copying runtime jar...
[info] [launcher] getting org.scala-sbt sbt 1.7.3 (this may take some time)...
[info] [launcher] getting Scala 2.12.17 (for sbt)...
[info] welcome to sbt 1.7.3 (N/A Java 17.0.4)
[info] loading project definition from /home/runner/avr-lista/project
[info] loading settings for project avr-lista from build.sbt ...
[info] set current project to scala-playground (in build file:/home/runner/avr-lista/
)
[info] sbt server started at local:///home/runner/.sbt/1.0/server/c826635152e01393211
d/sock
[info] started sbt server
[info] terminate the server with `shutdown`
> run
[warn] scala-xml_2.12-2.1.0.jar no longer exists at /home/runner/Scala/.cache/coursie
r/v1/https/repo1.maven.org/maven2/org/scala-lang/modules/scala-xml_2.12/2.1.0/scala-x
ml_2.12-2.1.0.jar
[info] compiling 1 Scala source to /home/runner/avr-lista/target/scala-2.12/classes .
..
[info] Non-compiled module 'compiler-bridge_2.12' for Scala 2.12.17. Compiling...
[info] Compilation completed in 26.816s.
[info] running MediaLista
Marcos Bruno P. Campos
A média da lista List(10.0, 20.0, 30.0, 40.0, 50.0) é 30.0
[success] Total time: 32 s, completed Jun 13, 2024, 6:55:29 PM

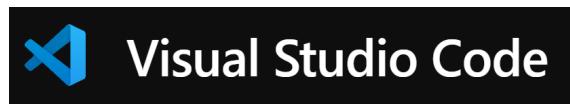
```

Figure 4.10: Media Lista Terminal

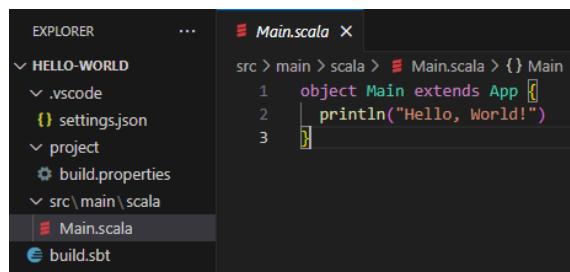
5. Ferramentas existentes e utilizadas

5.1 Editores para Scala

Vscode



- Visual Studio Code v1.9
- site oficial: <https://code.visualstudio.com>
- descrição: editor de código popular desenvolvido pela Microsoft
- Extensão Scala(Metals) v1.35.0



Replit



- Replit Versão Web
- site oficial: <https://replit.com>
- descrição: editor online de várias linguagens

 A screenshot of the Replit web interface. On the left, there's a file tree showing various files like 'Fibonacci.scala', 'Fatorial.txt', etc. The main area is a code editor with the following Scala code:


```

object Fibonacci {
  def fibonacci(n: Int) = n match {
    case 0 => 0
    case 1 => 1
    case _ => fibonacci(n - 1) + fibonacci(n - 2)
  }
}

def main(args: Array[String]): Unit = {
  val numero = 10
  println(s"A sequência de Fibonacci até $numero é:")
  for (i <- 0 to numero) {
    print(s"${fibonacci(i)} ")
  }
}
  
```

5.2 Compiladores

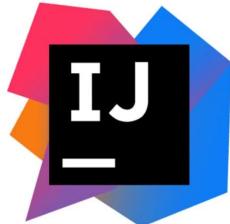


- Site principal : <https://www.scala-lang.org/>
- Scala 3 : <https://www.scala-lang.org/download/>

5.3 Ambientes de Programação IDE para Scala

Uma IDE, ou Ambiente de Desenvolvimento Integrado (do inglês Integrated Development Environment), é uma ferramenta de software que oferece recursos integrados para facilitar o desenvolvimento de software. Ela é projetada para ser um ambiente completo onde desenvolvedores podem escrever, testar, depurar e implementar código de maneira eficiente e produtiva.

Seguem 4 ótimas opções:



- IntelliJ IDEA <https://www.jetbrains.com/idea/>



- Eclipse <https://eclipseide.org/>



- NetBeans <https://netbeans.apache.org/front/main/index.html>
- Metals + Vscode (ver em: Ferramentas Utilizadas)



6. Considerações Finais

6.1 Conclusão

A linguagem Scala emerge como uma poderosa ferramenta no arsenal de qualquer desenvolvedor moderno, oferecendo uma combinação única de paradigmas funcional e orientado a objetos. Este livro oferece uma exploração abrangente dos conceitos fundamentais e avançados da Scala, destacando sua flexibilidade, expressividade e eficiência.

Ao longo das suas páginas, fica claro que Scala não é apenas uma extensão do Java, mas sim uma linguagem que abre novas possibilidades para desenvolvedores, permitindo a criação de sistemas robustos e escaláveis com um código mais conciso e legível. A integração perfeita com o ecossistema Java e sua interoperabilidade com outras linguagens tornam Scala uma escolha atraente para empresas que buscam modernizar suas aplicações sem abandonar o investimento em infraestrutura existente.

6.1.1 Problemas enfrentados

Durante o desenvolvimento deste trabalho, diversos problemas foram enfrentados. Entre os principais desafios estão:

- Ler diversos conteudos e corrigir erros no latex(ferramenta de criação desse livro)
- Implementar um ambiente em Scala que funcionasse de acordo com os requisitos dos programas
- Conciliar o desenvolvimento do livro com os diversos exercícios de paradigmas de programação

6.1.2 Resumo do trabalho desenvolvido

O trabalho desenvolvido pode ser resumido nas seguintes etapas:

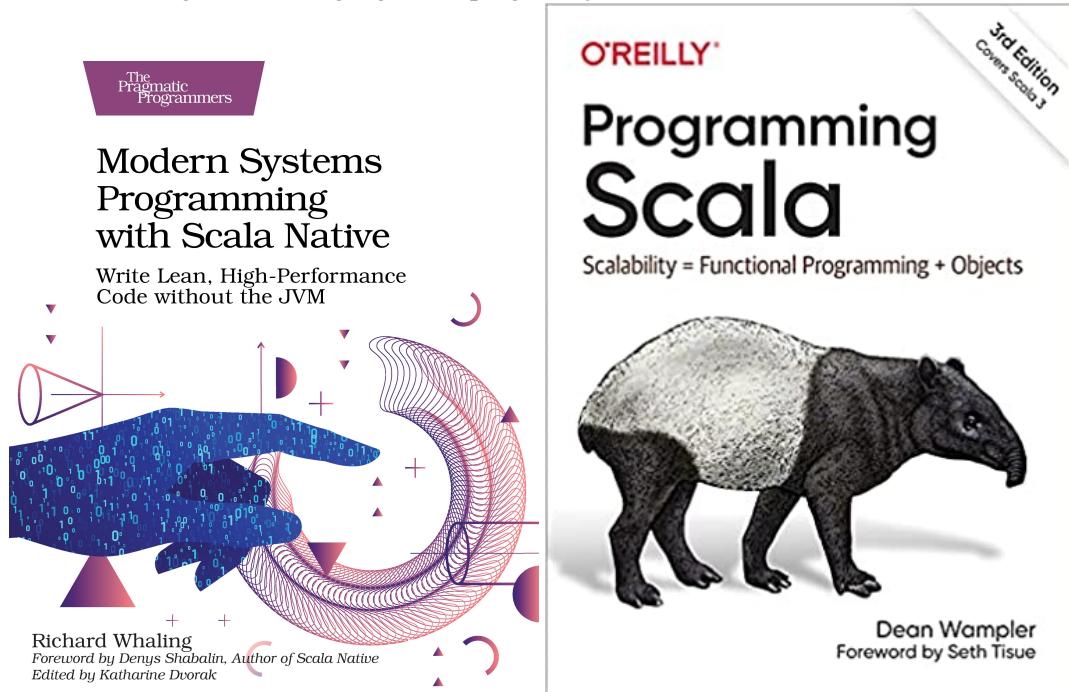
- Consulta de diversos livros em portugues e estrangeiros.
- Aprendizado de commandos em Latex
- Consultas a modelos IA
- Testes de Programas em Ambiente de programação
- Busca de dados para ilustrar os conceitos abordados

6.1.3 Aspectos não considerados e sugestões para trabalhos futuros

Alguns aspectos não foram considerados no presente trabalho, mas poderiam ser estudados ou serem úteis para pesquisas futuras:

Esse é um livro introdutorio, portanto muitos conceitos avançados foram deixados de lado, para um melhor entendimento da linguagem e conceitos do paradigma orientado a objeto recomendo ler os seguintes livros:

Figure 6.1: Linguagens de programação modernas e um bom livro



Fonte: O autor



Bibliography

- [BV09] Frank Sommers Bill Venners. The origins of scala a conversation with martin odersky. *artima*, 2009. Citado na página [5](#).
- [Wam21] Dean Wampler. *Programming Scala*. O'Reilly Media Inc., Sebastopol, CA, 3 edition, July 2021. Citado 3 vezes nas páginas [9](#), [10](#) e [17](#).

