Capítulo 2

Scala: recursos básicos

Scala[1] é uma linguagem de programação que se estrutura em dois paradigmas principais: a programação funcional e a programação orientada a objetos. Por ser multiparadigma, temos a possibilidade de criar programas mais fortemente orientados a objetos (*oo-first*) ou mais fortemente funcionais (*functional-first*). Dada a ênfase do nosso curso em programação funcional, vamos explorar mais a fundo os recursos funcionais da linguagem. No entanto, é importante ressaltar que Scala também possui uma forte base de programação orientada a objetos, e muitos dos conceitos que veremos são comuns a outras linguagens orientadas a objetos, como Java.

Características principais As diretrizes de projeto da linguagem Scala foram definidas para criar uma linguagem que combina as melhores características da programação funcional e da programação orientada a objetos. A seguir, listamos algumas das principais características que tornam Scala uma linguagem poderosa e versátil:

- **Sintaxe concisa**: Scala permite escrever código de forma mais expressiva e menos verbosa do que sua principal inspiração, Java. A sintaxe é projetada para ser mais limpa e legível, reduzindo a quantidade de código repetitivo (boilerplate) necessário.

- **Compilada**: Scala é uma linguagem compilada, ou seja, o código fonte é transformado em bytecode que pode ser executado na JVM (*Java Virtual Machine*). Há também a possibilidade de compilar o código Scala diretamente para JavaScript ou código nativo.

- **Programação funcional avançada**: Além dos recursos básicos oferecidos pela maioria das linguagens multiparadigma, Scala suporta conceitos avançados de programação funcional, como classes de tipos, mônadas, functores, functores aplicativos e monóides, que permitem uma programação mais expressiva e modular.

- **Tipagem estática**: Scala é uma linguagem estaticamente tipada, o que significa que os tipos de dados são verificados em tempo de compilação, ajudando a evitar erros comuns.

[1] `https://www.scala-lang.org/`

- **Inferência de tipos**: Scala possui um sistema avançado de inferência de tipos que permite ao compilador deduzir os tipos de variáveis e expressões, reduzindo a necessidade de declarações explícitas, o que também contribui para a concisão do código.

- **Imutabilidade por padrão**: As estruturas de dados da biblioteca-padrão de Scala são imutáveis por padrão, ou seja, uma vez criadas, não podem ser modificadas. Isso ajuda a evitar efeitos colaterais indesejados e torna o código mais previsível e fácil de entender. Além disso, a imutabilidade é um conceito central na programação funcional, pois permite que funções sejam aplicadas de forma segura e eficiente.

- **Concorrência e paralelismo**: Scala possui bibliotecas e construções que facilitam a programação concorrente e paralela, como o Akka.

- **Compatibilidade com Java**: Scala é totalmente interoperável com Java, permitindo que bibliotecas Java sejam usadas diretamente em código Scala.

2.1 Histórico

- **2001-2003**. Martin Odersky inicia o desenvolvimento da linguagem Scala na École Polytechnique Fédérale de Lausanne (EPFL), na Suíça. Odersky é conhecido por seu trabalho em linguagens de programação e compiladores, e já havia contribuído para o desenvolvimento, por exemplo, dos recursos de programação genérica da linguagem Java (*Java Generics*) e do compilador `javac`.

  O objetivo inicial era criar uma linguagem cuja sintaxe fosse mais concisa do que Java, mas que ainda fosse executada na JVM, permitindo a reutilização de bibliotecas Java existentes. Além disso, Odersky queria incorporar conceitos de programação funcional na linguagem, sem prejuízo aos recursos de orientação a objetos, algo que não era bem suportado em Java na época.

- **2004-2006**. A primeira versão pública do Scala é lançada, como uma linguagem que executa na Java Virtual Machine (JVM).

  Em 2006, a versão 2.0 do Scala é lançada, trazendo melhorias significativas na linguagem, incluindo suporte a inferência de tipos, que permite ao compilador deduzir os tipos de variáveis e expressões sem a necessidade de declarações explícitas. Essa versão também introduziu o conceito de *case classes*, que facilitam a criação de classes imutáveis com suporte a correspondência de padrões. Além de diferentes recursos para programação concorrente e paralela, como o *Actor Model*, que se tornou a fundação para o famoso framework Akka.

- **2006-2020**. Com seu foco em programação funcional e concorrência, forte integração com orientação a objetos e Java, Scala ganha popularidade entre grandes empresas de tecnologia, especialmente aquelas que trabalham com grandes volumes de dados e sistemas distribuídos. Empresas como Twitter, LinkedIn e Netflix adotam Scala para construir suas infraestruturas e serviços.

Scala torna-se a fundação para o desenvolvimento de frameworks populares, como o Akka e o Apache Spark, que é amplamente utilizado para processamento de grandes volumes de dados. O Akka, por exemplo, é um framework para construção de sistemas concorrentes e distribuídos, enquanto o Spark é um motor de processamento de dados em larga escala que permite o processamento distribuído de grandes conjuntos de dados.

- **2021-2025**. A versão 3.0 do Scala é lançada, trazendo grandes mudanças, incluindo uma sintaxe mais concisa e expressiva, além de melhorias no sistema de tipos. A versão 3.0 também introduz o conceito de *enumerações* e *match types*, que permitem uma correspondência de padrões mais poderosa e flexível. Além disso, a versão 3.0 traz melhorias na interoperabilidade com Java e outras linguagens, tornando Scala ainda mais acessível para desenvolvedores que já estão familiarizados com a JVM.

  Scala sofre concorrência com outras linguagens de programação para o desenvolvimento de backend, como Kotlin, Go e Java com bibliotecas, que também estão ganhando popularidade por suas características modernas e suporte a programação funcional. No entanto, Scala continua a ser uma linguagem poderosa e versátil, com uma comunidade ativa e um ecossistema rico de bibliotecas e frameworks. Entre os principais nichos em que Scala se destaca estão o desenvolvimento de sistemas distribuídos, processamento de dados em larga escala e aplicações que exigem alta concorrência e paralelismo. A linguagem continua a evoluir, com novas versões sendo lançadas regularmente, trazendo melhorias de desempenho, novos recursos e correções de bugs.

2.2 Ambiente de desenvolvimento

Para desenvolver em Scala, precisamos fundamentalmente das seguintes ferramentas:

- `scalac`: o compilador Scala, que converte o código fonte Scala em bytecode que pode ser executado na JVM.
- `scala`: ferramenta com múltiplos propósitos, que pode ser usada para:
  - Executar diretamente código Scala, no modo script.
  - Iniciar um REPL (*Read-Eval-Print Loop*), que é o ambiente interativo para experimentar com código Scala.

Compilação com `scalac`. Quando desenvolvemos em Scala, não é comum utilizar o compilador `scalac` diretamente, pois os artefatos gerados por essa ferramenta exigem um certo conhecimento da arquitetura da linguagem para serem executados. No entanto, para que tenhamos uma boa percepção de que Scala é uma linguagem compilada, vamos experimentar com um exemplo de compilação direta.

Como base para experimentação, a listagem 2.1 apresenta um programa mínimo em Scala 3.0 que imprime a mensagem `"Hello, World!"` no terminal.

Código 2.1: Exemplo de programa em Scala 3.0

```scala
@main def helloWorld() =
  println("Hello, World!")
```

Podemos compilar o código fonte acima usando o comando `scalac`:

```
$ scalac HelloWorld.scala
```

Como resultado do processo de compilação, o compilador Scala gera diferentes arquivos, dependendo do conteúdo do código fonte. Isso ocorre para compatibilizar o código Scala com a JVM.

Execução do código compilado. Devido à variedade de arquivos gerados por `scalac`, o processo de execução do código gerado não é muito intuitivo. Para executar o código compilado, utilizamos o comando `scala` seguido do nome da classe que contém o ponto de entrada do programa — no nosso caso, é a função anotada com `@main`. Para o exemplo apresentado, o comando seria:

```
$ scala helloWorld
Hello, World!
```

Compilação e execução integradas com `scala` A ferramenta `scala` simplifica bastante o processo de compilação e execução do código. Ao invés de compilar o código fonte separadamente com `scalac` e depois executar com `scala`, podemos fazer tudo em um único passo.

Para isso, basta executar o comando `scala` seguido do nome do arquivo que contém o código fonte. O comando irá compilar o código e, em seguida, executá-lo. Por exemplo:

```
$ scala HelloWorld.scala
Hello, World!
```

Esse é um dos métodos preferidos para executar código Scala, especialmente durante o desenvolvimento e testes, pois simplifica o processo e permite uma iteração mais rápida.

REPL O REPL (*Read-Eval-Print Loop*) é uma ferramenta que permite executar código Scala incremental e interativamente, sem a necessidade de criar arquivos de código fonte ou compilar o código. O REPL é especialmente útil para experimentar com pequenos trechos de código, testar expressões e aprender a linguagem. Para iniciar o REPL, basta executar o comando `scala` sem argumentos:

```
$ scala
Welcome to Scala 3.3.6 (21.0.7, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala>
```

No REPL, você pode digitar expressões Scala e pressionar Enter para avaliá-las. Por exemplo:

```
scala> "Hello, world!"
val res0: String = Hello, world!
scala> 1 + 2
```

```
res0: Int = 3
scala> val x = 42
x: Int = 42
scala> x * 2
res1: Int = 84
```

2.3 Estrutura de um programa Scala

A estrutura de um programa Scala pode seguir dois modelos principais: o modelo orientado a objetos e o modelo funcional. O modelo orientado a objetos foi predominante até as versões 2.x da linguagem. Já o modelo funcional tornou-se mais comum a partir das versões 3.x.

Estrutura orientada a objetos (Scala 2.x). No modelo orientado a objetos, um programa Scala é composto por uma ou mais classes, as quais, por sua vez, podem conter métodos e propriedades. Esse modelo é similar ao utilizado na linguagem Java, na qual a estrutura básica de um programa é composta por uma classe principal que contém o ponto de entrada do programa, geralmente um método chamado `main`. O exemplo de código 2.2 apresenta um exemplo de estrutura orientada a objetos em Scala.

Código 2.2: Exemplo de estrutura orientada a objetos

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

Os requisitos fundamentais para a estrutura orientada a objetos são:

- **Singleton**. Todo programa Scala deve conter pelo menos um `object`, contendo um método `main`. Um *object* em Scala é um *singleton*, ou seja, é uma classe que só pode ser instanciada uma vez. O ambiente de execução Scala gera automaticamente a instância implícita de um `object` — não é necessário criar uma instância explicitamente.
- **Método principal**. O método principal é o ponto de entrada do programa, onde a execução começa. Ele deve ser definido dentro de um `object` e deve ter a assinatura `def main(args: Array[String])`. Caso a assinatura do método principal não seja respeitada, o ambiente de execução não reconhecerá o ponto de entrada do programa, resultando em um erro de execução.

Delimitação de blocos A delimitação de blocos de código em Scala pode ser feita de dois modos:

- **Chaves (Scala 2.x)**. Utilizamos chaves para definir blocos de código, como em linguagens como Java e C++. Por exemplo, o exemplo de código 2.2 pode ser reescrito utilizando chaves para delimitar os blocos do `object` `HelloWorldBraces` e do método `main`, conforme o exemplo de código 2.3.

Código 2.3: Exemplo de estrutura orientada a objetos com chaves

```scala
object HelloWorldBraces {
  def main(args: Array[String]) = {
      println("Hello, World with Braces!")
  }
}
```

- **Indentação (Scala 3.x).** A partir da versão 3.0, Scala introduziu uma nova sintaxe que permite definir blocos de código utilizando apenas indentação, similar ao Python. Essa abordagem torna o código mais legível e reduz a necessidade de chaves. Porém, para que essa abordagem funcione, é necessário que a quantidade de espaços em branco seja consistente em todo o bloco de código, e que não se misture espaços com tabulações. A listagem 2.2 é um exemplo dessa abordagem. Quando seguimos o modelo de indentação, a declaração das classes e objetos deve terminar com dois pontos (`:`).

Estrutura funcional (Scala 3.x). No modelo funcional, um programa Scala é composto por uma ou mais funções no nível mais alto do programa, ou seja, não requer a definição de classes ou `objects`. O exemplo de código 2.4 apresenta um exemplo de estrutura funcional em Scala.

Código 2.4: Exemplo de estrutura funcional

```scala
def sayName = "John Doe"

def getAge(yearOfBirth: Int) =
    java.time.Year.now.getValue - yearOfBirth

@main def main() =
    println("Name: " + sayName)
    println("Age: " + getAge(2000))
```

```
Name: John Doe
Age: 25
```

O programa 2.4 define três funções no nível mais alto:

- `sayName`, que retorna uma string com o nome do usuário.
- `getAge`, que recebe um parâmetro `yearOfBirth` e calcula a idade do usuário com base no ano atual.
- `main`, que é o ponto de entrada do programa, onde as outras funções são chamadas e os resultados são impressos no console.

Note que não precisamos definir essas funções como métodos de uma classe ou `object`, pois estamos seguindo o modelo funcional. A função `main` deve ser anotada com `@main`, indicando que é o ponto de entrada do programa, e não precisa ser definida dentro de um `object`.

FUNÇÕES Um programa Scala precisa ter ao menos uma função, que deve ser o ponto de entrada do programa. Como vimos anteriormente, essa função pode ser o método `main` de um `object` no modelo orientado a objetos, ou uma função anotada com `@main` no modelo funcional. Em seções e capítulos posteriores, estudaremos mais a fundo aspectos importantes de funções em Scala, mas por enquanto, basta saber que uma função segue a sintaxe `def nome(parametros): tipo_de_retorno = corpo`, onde:

- `def` é a palavra-chave que indica a definição de uma função.
- `nome` é o nome da função, que deve ser um identificador válido em Scala.
- `parametros` são os parâmetros da função, que podem ser opcionais. Cada parâmetro deve ter um nome e um tipo, separados por dois pontos (`:`).
- `tipo_de_retorno` é o tipo de dado que a função retorna, também separado por dois pontos (`:`). Se a função não retornar um valor, esse tipo pode ser omitido.
- `corpo` é o código que será executado quando a função for chamada. O corpo pode ser uma expressão única ou um bloco de código.

2.4 Variáveis

Para declarar variáveis em Scala, precisamos de um modificador de mutabilidade, do nome da variável e de um valor inicial. Scala permite a declaração de variáveis mutáveis ou imutáveis, cada uma com um modificador específico:

- **Mutáveis (`var`).** Variáveis mutáveis devem ser declaradas com o modificador `var`, que indica que o valor da variável pode ser alterado após sua declaração. Utilizamos variáveis mutáveis quando nosso programa segue o estilo imperativo. A listagem 2.5 apresenta exemplos de variáveis mutáveis.

Código 2.5: Exemplo de variáveis mutáveis

```
var x = 10

x = 20
println(x)

var y = "Hello"

y = "World"
println(y)
```

```
20
World
```

- **Imutáveis (`val`).** Variáveis imutáveis são declaradas com o modificador `val`, que indica que o valor da variável não pode ser alterado após sua declaração.

Utilizamos variáveis imutáveis quando nosso programa segue o estilo funcional ou declarativo. A listagem 2.6 apresenta um exemplo de variável imutável.

Código 2.6: Exemplo de variável imutável

```scala
val x = 10

x = 20 // Compilation error
println(x)
```

```
-- [E052] Type Error: /workspaces/dev-env-tutorial/src/Variables.scala:3:6
   -----
3 | x = 20 // Compilation error
  | ^^^^^^
  | Reassignment to val x
  |
  | longer explanation available when compiling with `-explain`
1 error found
Errors encountered during compilation
```

Pureza funcional e imutabilidade. Na programação funcional pura, todas as variáveis devem ser imutáveis, ou seja, declaradas com `val`. Isso garante que o estado do programa não mude inesperadamente, tornando o código mais previsível e fácil de entender. No entanto, em Scala, é possível utilizar variáveis mutáveis (`var`) quando necessário, especialmente em contextos imperativos ou quando a performance é uma preocupação. Quando queremos atingir pureza funcional, devemos declarar todas as variáveis como imutáveis (`val`), além de tomar medidas adicionais, como evitar efeitos colaterais e garantir que as funções sejam puras, como veremos em seções posteriores.

2.5 Tipos de dados

Tipagem estática. Scala é uma linguagem estaticamente tipada, o que significa que:

- as variáveis devem ser declaradas com um tipo específico, que se mantém durante todo o tempo de vida da variável;
- o tipo de uma variável é verificado em tempo de compilação, o que ajuda a evitar erros comuns relacionados a tipos de dados.

A listagem 2.7 apresenta exemplos de variáveis com tipos explícitos.

Código 2.7: Exemplo de variáveis com tipos explícitos

```scala
val x: Int = 10
val y: String = "Hello, World!"
val z: Boolean = true
```

```scala
val pi: Double = 3.14
val list: List[Int] = List(1, 2, 3, 4, 5)
val map: Map[String, Int] = Map("one" -> 1, "two" -> 2, "three" -> 3)

List(x, y, z, pi, list, map)
        .foreach(item => println(s"${item.getClass.getSimpleName}: $item"))
```

```
Int: 10
String: Hello, World!
Boolean: true
Double: 3.14
List: List(1, 2, 3, 4, 5)
Map: Map(one -> 1, two -> 2, three -> 3)
```

LITERAIS. Scala suporta diferentes tipos de literais, que são representações fixas de valores no código fonte. A tabela 2.1 apresenta alguns exemplos de literais em Scala.

Tipo	Exemplo
Inteiro	`42`
Ponto flutuante	`3.14`
String	`"Hello, World!"`
Booleano	`true`
Tupla	`(1, "dois", true)`
Lista	`List(1, 2, 3)`
Conjunto	`Set(1, 2, 3)`
Mapa	`Map("um" -> 1, "dois" -> 2)`
Função	`(x: Int) => x + 1`

Tabela 2.1: Exemplos de literais em Scala.

INFERÊNCIA DE TIPOS. Scala possui um sistema avançado de inferência de tipos, o que significa que o compilador pode deduzir automaticamente o tipo de uma variável com base no valor atribuído a ela. Isso permite que os desenvolvedores escrevam código mais conciso e legível, sem a necessidade de declarar explicitamente os tipos em muitos casos. A listagem 2.8 apresenta exemplos de variáveis com tipos inferidos.

Código 2.8: Exemplo de variáveis com tipos inferidos

```scala
val x = 10
val y = "Hello, World!"
val z = true
val pi = 3.14
val list = List(1, 2, 3, 4, 5)
val map = Map("one" -> 1, "two" -> 2, "three" -> 3)

List(x, y, z, pi, list, map)
    .foreach(item => println(s"${item.getClass.getSimpleName}: $item"))
```

```
Int: 10
String: Hello, World!
Boolean: true
Double: 3.14
List: List(1, 2, 3, 4, 5)
Map: Map(one -> 1, two -> 2, three -> 3)
```

TIPOS COMO OBJETOS. Em Scala, todos os tipos de dados são tratados como classes, ou seja, não temos tipos primitivos como em Java ou C++. A figura 2.1 mostra um diagrama simplificado da hierarquia de tipos em Scala.

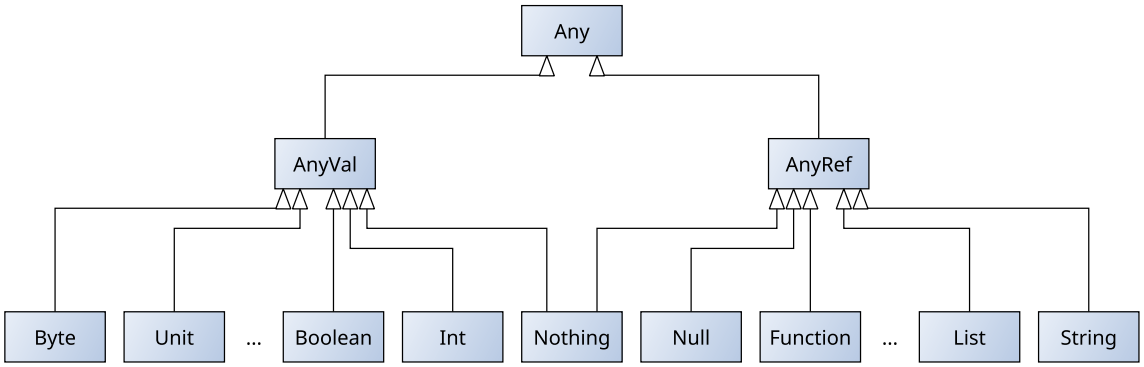

Figura 2.1: Diagrama simplificado da hierarquia de tipos em Scala.

O tipo universal `Any` é a superclasse de todos os tipos em Scala. Todos os tipos de dados, incluindo os que tradicionalmente são definidos como primitivos em outras linguagens, são subclasses de `Any`. A partir do tipo `Any`, temos duas subclasses principais: `AnyVal` e `AnyRef`.

- **`AnyVal`**. Representa os tipos de valor, que são tipos primitivos em outras linguagens, como `Int`, `Double`, `Boolean`, etc. Esses tipos são todos imutáveis. Embora sejam representados como objetos em Scala, eles são convertidos para tipos primitivos quando o código é transformado em bytecode para a JVM. Isso significa que, apesar de serem tratados como objetos, na prática eles não têm o mesmo *overhead* de memória e desempenho que os tipos de referência. O melhor desempenho deriva do fato de que, na JVM, os tipos primitivos têm tamanho fixo e podem ser armazenados diretamente na pilha de execução.

- **`AnyRef`**. Representa os tipos de referência, ou seja, aqueles que na JVM serão tratados como objetos. Esses tipos, quando compilados para a JVM, são armazenados na heap, podendo ser mutáveis ou imutáveis, a depender de como foram definidos. Tipos pré-definidos na linguagem incluem `String`, `List`, `Map`, etc. Tipos customizados pelo usuário, definidos como classes e `objects`, também são subclasses de `AnyRef`.

Além dos dois tipos principais, `AnyVal` e `AnyRef`, Scala também possui outros tipos de dados, como:

- **`Unit`**. Representa a ausência de valor, similar ao `void` em Java. É o tipo de retorno padrão para "funções" que não retornam um valor significativo. Na teoria de linguagens de programação, há diferenciação entre funções e procedimentos: funções sempre retornam um valor, enquanto procedimentos não. O tipo `Unit` é usado para indicar que a unidade de execução é um procedimento, e não uma função. Sempre que definimos um procedimento, estamos quase sempre operando no paradigma imperativo, pois provavelmente estamos gerando efeitos colaterais.

- **`Null`**. Representa a ausência de um valor de referência, ou seja, uma referência que não aponta para nenhum objeto. É um subtipo de `AnyRef`. Em programação funcional pura, devemos evitar o uso desse valor: sua presença se justifica por questões de compatibilidade com código em Java.

- **`Function`**. Representa funções como objetos, permitindo que sejam tratadas como cidadãos de primeira classe. Scala possui diferentes tipos de função, dependendo do número de parâmetros e do tipo de retorno, como `Function1`, `Function2`, etc.

- **`Nothing`**. Representa um tipo que não tem valores, ou seja, é um tipo vazio. É usado para indicar que uma função nunca retorna (por exemplo, uma função que lança uma exceção).

- etc.

2.6 Expressões

Expressões são trechos de código que, quando avaliadas, resultam em um valor. Assim, expressões podem ser usadas em qualquer ponto do programa onde um valor é esperado. Em linguagens funcionais, existe uma grande variedade de construções que podem ser consideradas expressões. De fato, numa linguagem puramente funcional, tudo é expressão: o programa em si pode ser visto como uma expressão, formada por uma composição de expressões. Devido a essa amplitude de tipos de expressões, vamos de início focar em algumas mais simples, para ampliarmos o conceito posteriormente.

Expressões aritméticas. A notação de expressões aritméticas é semelhante à das principais linguagens de programação. A tabela 2.2 mostra exemplos de expressões envolvendo os principais operadores aritméticos.

Operador	Exemplo
Adição	`1 + 2`
Subtração	`2 - 1`
Multiplicação	`2 * 3`
Divisão	`2 / 3`
Resto da divisão	`2 % 3`

Tabela 2.2: Exemplos de expressões aritméticas em Scala.

OPERADORES VS. MÉTODOS Os operadores na notação infixa, como os apresentados na tabela 2.2, são na verdade açúcar sintático para os métodos definidos nas classes dos tipos numéricos. Por exemplo, a expressão `1 + 2` é equivalente a `1.+(2)`. Isso significa que podemos: i) definir nossos próprios operadores, que não existem na linguagem, e ii) sobrescrever o comportamento dos operadores pré-definidos para atribuir uma semântica diferente da original.

EXPRESSÕES LÓGICAS. As expressões lógicas são usadas para combinar valores booleanos e realizar operações lógicas. A tabela 2.3 mostra exemplos de expressões lógicas em Scala.

Operador	Exemplo
E lógico (AND)	`true && false`
Ou lógico (OR)	`true || false`
Negação lógica (NOT)	`!true`

Tabela 2.3: Exemplos de expressões lógicas em Scala.

EXPRESSÕES RELACIONAIS. As expressões relacionais são usadas para comparar valores e determinar relações entre eles. Elas retornam um valor booleano (`true` ou `false`) com base na comparação. A tabela 2.4 mostra exemplos de expressões relacionais em Scala.

Operador	Exemplo
Igualdade	`1 == 1`
Desigualdade	`1 != 2`
Maior que	`2 > 1`
Menor que	`1 < 2`
Maior ou igual a	`2 >= 1`
Menor ou igual a	`1 <= 2`

Tabela 2.4: Exemplos de expressões relacionais em Scala.

EXPRESSÕES DE BLOCO Scala possui uma construção muito peculiar, na qual podemos definir um bloco de expressões que, quando avaliado, resulta em um valor. Por exemplo, considere o exemplo de código 2.9. Note que o que define o bloco é a indentação.

Código 2.9: Exemplo de expressão de bloco

```scala
val result =
  val x = 10
  val y = 20
  x + y // The last expression in the block is the result

println(result)
```

```
30
```

Nesse exemplo, o bloco de código indentado é uma expressão composta que define duas variáveis locais (`x` e `y`) e, em seguida, calcula a soma entre elas. O valor da última expressão no bloco (`x + y`) é o valor retornado pela expressão de bloco, que é atribuído à variável `result`.

Expressões de bloco são úteis para agrupar operações e retornar um único valor, tornando o código mais organizado e legível. Elas podem ser usadas em qualquer lugar onde uma expressão é esperada, como na atribuição de variáveis, em argumentos de funções ou em estruturas de controle de fluxo.

2.7 Controle de fluxo

A noção de controle de fluxo é fundamental em qualquer linguagem de programação, pois permite que o programa tome decisões e execute diferentes blocos de código conforme condições específicas. O modo como o controle de fluxo ocorre depende do paradigma de programação adotado.

- No **paradigma imperativo**, o controle de fluxo é determinado pela execução sequencial de comandos e por desvios condicionais e saltos. A execução sequencial é a ordem natural de execução dos comandos, enquanto desvios condicionais e saltos podem ser controlados por estruturas específicas, como `if` para seleção, `while` e `for` para repetição. As estruturas de repetição, em particular, dependem fortemente de mutabilidade para operar.
- No **paradigma funcional**, o controle de fluxo é baseado na avaliação de expressões e na aplicação de funções. Na programação funcional, a ordem de execução das ações é determinada pela ordem de avaliação das expressões e pela composição de funções. Para obter esse comportamento, não é necessário utilizar mutabilidade nem estruturas de controle.

Como Scala é multiparadigma, oferece suporte a ambos os tipos de controle de fluxo. Neste capítulo, focaremos principalmente no controle de fluxo funcional e abordaremos apenas os recursos relacionados a esse paradigma.

Expressões condicionais. Diferentemente da programação estruturada, em que a estrutura `if` é um comando, em Scala ela é uma expressão, ou seja, sempre retorna um valor. Por exemplo, considere as expressões condicionais apresentadas na listagem 2.10.

Código 2.10: Exemplo de expressões condicionais

```scala
val x = 10
val y = 20

val max = if (x > y) x else y
println(s"Max of $x and $y is: $max")

val isEven = if (x % 2 == 0) "even" else "odd"
println(s"$x is $isEven")

```

```scala
// an example of if expression with blocks without side effects
val totalItems = 12

val price = if (totalItems > 10)
    val pricePerItem = 100
    val tax = 0.1

    totalItems * pricePerItem * (1 + tax)
else
    val pricePerItem = 150
    val comission = 0.2

    totalItems * pricePerItem * (1 + comission)

println(s"Total price for $totalItems items is: $price")
```

```
Max of 10 and 20 is: 20
10 is even
Total price for 12 items is: 1320.0
```

Podemos comparar a expressão condicional `if` ao operador ternário de outras linguagens, como Java e C++. De fato, uma expressão `if` exerce o mesmo papel do operador ternário, mas com a vantagem de que podemos utilizar blocos de código mais complexos, com múltiplas expressões, sem a necessidade de utilizar parênteses para delimitar o escopo da expressão. Isso torna a sintaxe mais gerenciável e legível.

Repetição declarativa Nas linguagens imperativas, a repetição é geralmente implementada com estruturas de controle como `for`, `while` e `do-while`. No entanto, na programação funcional, a repetição é tratada de forma diferente, utilizando diferentes recursos, como:

- **Recursão**. A recursão é a técnica fundamental para implementar repetição na programação funcional. Modelamos em o processo repetitivo em caso(s)-base e caso(s)-recursivo(s).
- **Funções de ordem superior**. Funções como `map`, `filter` e `fold` permitem aplicar uma função a cada elemento de uma coleção, filtrando ou reduzindo os resultados de forma declarativa. Essas funções são frequentemente usadas para iterar sobre coleções sem a necessidade de estruturas de controle explícitas. Nesse caso, temos um proceso de repetição centrado em dados.
- **Compreensões de coleção**. Muitas linguagens funcionais, incluindo Scala, oferecem uma sintaxe especial para criar coleções de forma declarativa, permitindo expressar operações de repetição e transformação de forma concisa e legível. As compreensões de coleção (*comprehensions*) são uma forma poderosa de expressar operações em coleções, como filtragem, mapeamento e redução, sem a necessidade de estruturas de controle explícitas.

Os recursos de repetição declarativa são um tópico extenso que merece um estudo mais aprofundado, que faremos adiante. Por enquanto, basta saber que, na

programação funcional pura, não devemos recorrer a estruturas de controle imperativas, como `for` e `while`, para implementar repetição. Em vez disso, devemos utilizar recursos como recursão, funções de ordem superior e compreensões de coleção.

2.8 Saída de dados

A saída de dados em Scala é feita principalmente através da função `println`, que imprime uma mensagem no console. Essa função é uma das mais comuns e úteis para depuração e exibição de informações durante o desenvolvimento. A listagem 2.11 apresenta um exemplo simples de uso da função `println`.

Código 2.11: Exemplo de saída de dados

```
println("Hello, World!")
println("This is a simple Scala program.")
```

INTERPOLAÇÃO DE STRINGS. A função `println` aceita diferentes tipos de argumentos, incluindo strings, números e outros tipos de dados. Além disso, podemos formatar previamente a string enviada para a saída usando interpolação de strings, que é uma maneira conveniente de incluir valores de variáveis dentro de strings. A listagem 2.12 apresenta um exemplo de interpolação de strings.

Código 2.12: Exemplo de formatação de saída

```
val name = "John Doe"
val age = 30

println(s"Name: $name, Age: $age")
```

```
Name: John Doe, Age: 30
```

Quando precisamos aplicar uma função dentro de uma interpolação de string, utilizamos a sintaxe `${}`. Por exemplo, considere a listagem 2.13.

Código 2.13: Exemplo de formatação de saída com aplicação de função

```
println(s"Random number: ${scala.util.Random.nextInt(100)}")
```

```
Random number: 42
```

INTERPOLADORES Os exemplos das listagens 2.12 e 2.13 utilizam o interpolador `s`, que é o mais comum e permite incluir variáveis diretamente na string. O interpolador `f` permite formatar valores de forma mais precisa, semelhante ao método `printf` em outras linguagens. Por exemplo, podemos formatar números com casas decimais específicas ou aplicar formatação de moeda. A listagem 2.14 apresenta um exemplo de uso do interpolador `f`.

Código 2.14: Exemplo de formatação de número

```
println(f"Sine: ${math.sin(math.Pi / 2)}%.2f")
println(f"Value of pi: ${math.Pi}%.4f")
```

```
Sine: 1.00
Value of pi: 3.1416
```

Uma alternativa ao interpolador `f` é a função `printf`, que permite formatar strings de forma semelhante ao método `printf` em Java. Consulte a documentação oficial para mais detalhes sobre os interpoladores disponíveis e suas funcionalidades.

2.9 Listas

Scala possui uma rica biblioteca de estruturas de dados, chamada biblioteca-padrão *Collections*. As estruturas dessa biblioteca são divididas em dois grupos principais:

- **Imutáveis**. As coleções imutáveis são aquelas cujo conteúdo não pode ser alterado após sua criação. Qualquer operação que modifique uma coleção imutável resulta em uma nova coleção, preservando a original. Esse é o padrão recomendado na programação funcional, pois promove a segurança e a previsibilidade do código.

  Exemplos: `List`, `Set`, `Map`, `Vector`, etc.

- **Mutáveis**. As coleções mutáveis são aquelas cujo conteúdo pode ser alterado após sua criação. Elas permitem operações como inserção, remoção e modificação de elementos diretamente na coleção original, sem criar uma nova instância. Essas coleções são mais comuns em programação imperativa.

  Exemplos: `Array`, `ArrayBuffer`, `HashSet`, `HashMap`, etc.

Cada uma das estruturas possui uma API rica para manipulação de dados. Vamos explorar as estruturas incrementalmente ao longo do curso — com ênfase nas imutáveis, que são mais comuns na programação funcional. Neste primeiro momento, vamos aprender as operações básicas sobre a estrutura `List`, uma das mais utilizadas em Scala.

Características de `List` A estrutura `List` possui as seguintes características:

- **Imutável**. Os elementos da lista devem ser informados no momento da criação, e qualquer operação que modifique a lista resulta em uma nova lista, preservando a original. Essa propriedade é denominada *persistência*, pois a lista original permanece inalterada, e uma nova lista é criada com as modificações desejadas.

- **Encadeamento simples**. A implementação da lista é baseada em encadeamento simples, em que cada elemento aponta para o próximo. Isso significa

que a estrutura não possui acesso aleatório; o acesso a um elemento tem custo linear ($O(n)$) em relação ao tamanho da lista. Por outro lado, a inserção e remoção de elementos no início da lista são operações eficientes, com custo constante ($O(1)$).

- **Homogênea**. Todos os elementos da lista devem ser do do tipo genérico que ela especifica, ou seja, a lista é homogênea. Ao criar uma lista, devemos especificar o tipo dos elementos que ela conterá. Para isso, utilizamos programação genérica. Por exemplo, podemos ter uma lista de inteiros (`List[Int]`), uma lista de strings (`List[String]`), ou mesmo uma lista que aceite qualquer tipo de dado (`List[Any]`).

Criação Para criar uma lista em Scala, utilizamos a sintaxe de literais de lista, que é uma sequência de elementos separados por vírgulas e envoltos por parênteses. A listagem 2.15 apresenta exemplos de criação de listas.

Código 2.15: Exemplo de criação de listas

```scala
val emptyList = List() // Lista vazia
val numbers = List(1, 2, 3, 4, 5) // Lista de inteiros
val names = List("Alice", "Bob", "Charlie") // Lista de strings
val mixed = List(1, "two", 3.0) // Lista mista (heterogênea)
```

Note que, devido à inferência de tipos, não é necessário especificar o tipo da lista ao criá-la, pois o compilador deduz automaticamente o tipo com base nos elementos. Em alguns casos, a inferência é um pouco mais sofisticada. Por exemplo, a lista `mixed` contém inteiros, reais e strings, tornando-a heterogênea. Nesse caso, o compilador deduz um tipo por união, como `List[Int | String | Double]`.

Lista vazia A lista vazia é uma lista que não contém nenhum elemento. Ela é representada por `List()` ou `Nil`. A listagem 2.16 apresenta exemplos de criação de listas vazias.

Código 2.16: Exemplo de lista vazia

```scala
val emptyList1 = List() // Lista vazia
val emptyList2 = Nil // Outra forma de representar a lista vazia
println(s"Empty list 1: $emptyList1")
println(s"Empty list 2: $emptyList2")
```

```
Empty list 1: List()
Empty list 2: List()
```

Operador de construção Podemos construir novas listas a partir de listas existentes utilizando o operador de construção `::`, que cria uma nova lista adicionando um elemento no início de uma lista existente. Esse operador é chamado de *cons operator* e é uma das principais características da estrutura de listas em Scala. A operação *cons* foi proposta inicialmente na linguagem Lisp e, devido à sua importância, foi adotada por várias outras linguagens funcionais, incluindo Scala.

O operador *cons* recebe dois operandos: o elemento a ser adicionado e a lista existente. O resultado é uma nova lista com o elemento adicionado no início, preservando a lista original. A listagem 2.17 apresenta exemplos de uso do operador `::`.

Código 2.17: Exemplo de uso do operador de construção

```
val numbers = List(2, 3, 4)
val newList = 1 :: numbers // Adiciona 1 no início da lista
println(newList) // Imprime: List(1, 2, 3, 4)
val anotherList = 0 :: 1 :: numbers // Adiciona 0 e 1 no início da lista
println(anotherList) // Imprime: List(0, 1, 2, 3, 4)
```

```
List(1, 2, 3, 4)
List(0, 1, 2, 3, 4)
```

Perceba que o operador `::` preserva a imutabilidade da lista original, criando uma nova lista com o elemento adicionado no início. O operador `::` é associativo à direita, o que significa que podemos encadear várias operações de construção de listas. Por exemplo, `1 :: 2 :: 3 :: Nil` cria uma lista com os elementos 1, 2 e 3.

Pelo fato de serem imutáveis, processos que constroem listas incrementalmente necessariamente geram listas intermediárias até a obtenção da lista final. Em geral, isso é feito por meio de processos recursivos, como demonstrado no exemplo de código 2.18, que constrói uma lista de inteiros de 1 a 10 em ordem reversa.

Código 2.18: Exemplo de construção recursiva de lista

```
def buildList(n: Int): List[Int] = {
  if (n <= 0) List() // Caso base: lista vazia
  else n :: buildList(n - 1) // Caso recursivo: adiciona n no início da lista
}

@main def main(): Unit = {
  val list = buildList(10)
  println(s"List from 1 to 10: $list")
}
```

```
List from 1 to 10: List(10, 9, 8, 7, 6, 5, 4, 3, 2, 1)
```

Decomposição Podemos decompor uma lista em seu primeiro elemento (cabeça ou *head*) e o restante da lista (cauda ou *tail*), utilizando os métodos `head` e `tail`, que retornam, respectivamente, o primeiro elemento e o restante da lista. Em Scala, métodos que não recebem argumentos atuam como propriedades e devem ser chamados sem utilizar parênteses. A listagem 2.19 apresenta um exemplo de uso dessas propriedades.

Código 2.19: Exemplo de uso de `head` e `tail`

```
val numbers = List(1, 2, 3, 4, 5)
```

```
println(s"Head: ${numbers.head}, Tail: ${numbers.tail}") // Imprime: Head: 1, Tail: List(2, 3, 4, 5)
```

```
Head: 1, Tail: List(2, 3, 4, 5)
```

Outros tópicos Há muita discussão ainda sobre o processamento de listas na programação funcional, pois essa é uma das principais estruturas de dados utilizadas nesse paradigma. Posteriormente, vamos explorar tópicos como funções de ordem superior, recursão e casamento de padrões, fundamentais para manipular listas de forma eficiente e expressiva.

2.10 Resumo

Scala é uma linguagem multiparadigma, integrando programação funcional e orientação a objetos. Destaca-se por características como sintaxe concisa, compilação para a JVM, além de opções para JavaScript e código nativo, suporte robusto à programação funcional, tipagem estática orientada a objetos com inferência de tipos, imutabilidade por padrão e interoperabilidade com Java.

O ambiente de desenvolvimento envolve o uso do compilador `scalac` e da ferramenta `scala` (para execução e REPL). Programas em Scala podem ser estruturados de diferentes formas, seja no estilo orientado a objetos (com `object` e método `main`) ou funcional (com funções de nível superior e anotação `@main`), permitindo a delimitação de blocos tanto por chaves quanto por indentação.

Variáveis podem ser declaradas como imutáveis (`val`) ou mutáveis (`var`), com preferência pela imutabilidade na abordagem funcional. O sistema de tipagem estática é orientado a objetos e conta com inferência de tipos.

Do ponto de vista do controle de fluxo, a linguagem suporta tanto o paradigma imperativo quanto o funcional, com ênfase em expressões condicionais (`if` como expressão) e recursos de repetição declarativa (recursão, funções de ordem superior e compreensões de listas). A saída de dados é realizada principalmente com `println`, que permite interpolação de strings para facilitar a formatação. A biblioteca-padrão de coleções oferece estruturas de dados imutáveis e mutáveis. Destaca-se a `List` (imutável, encadeamento simples, homogênea), que suporta criação com literais, o operador `::` (cons) e decomposição com `head` e `tail`.

2.11 Exercícios

Motivação. Esses exercícios visam cobrir os conteúdos abordados tanto no capítulo 2 quanto neste capítulo. São exercícios simples que visam reforçar sua familiaridade com a linguagem e com o estilo de programação funcional. Na seção 2.12, apresentamos as soluções para esses exercícios.

Restrições. A solução dos exercícios deve ser feita de modo estritamente funcional. Não é permitido o uso de estruturas de controle imperativas, como `for` e `while`. Além disso, não é permitido o uso de variáveis mutáveis (`var`). As soluções devem ser escritas utilizando apenas funções puras, sem efeitos colaterais.

Ex. 3.1 Defina uma função que receba uma nota $N \in [0, 10]$ e retorne uma string contendo o conceito correspondente, conforme a tabela a seguir:

Nota	Conceito
0 a 5.0	F
5.1 a 6.9	D
7.0 a 8.9	C
9.0 a 9.9	B
10.0	A

Ex. 3.2 Defina uma função que receba um inteiro, correspondente a uma qauntidade de produtos, e retorne a comissão de um vendedor de acordo com as regras da empresa:

Uma empresa paga a seus funcionários R$1,00 de comissão para cada produto vendido, entretanto, se forem vendidos mais de 250 produtos, o valor aumenta para R$1,50. Se a quantidade for superior a 500 produtos, o valor da comissão sobe para R$2,00. Calcule a comissão do funcionário com base na quantidade de produtos vendidos. Atenção: o cálculo deve ser sempre com base na maior comissão, ou seja, se o funcionário vender 600 produtos, ele deve receber R$2,00 por cada um dos produtos vendidos.

Ex. 3.3 Defina uma função que receba um número real e classifique a temperatura (em Celsius) de acordo com a tabela a seguir. Note que alguns intervalos têm intersecção. Por exemplo, se a temperatura for igual a `10.0`, ela se encaixa tanto na classificação *Frio* quanto na classificação *Fresco*. Por isso, a função deve retornar uma lista com todas as classificações possíveis. Caso haja mais de uma classificação, a ordem das classificações na lista deve ser a mesma da tabela.

Temperatura	Classificação
Inferior a -10	Congelante
-10 a 0	Muito Frio
0 a 10	Frio
10 a 20	Fresco
20 a 30	Agradável
30 a 40	Quente
Superior a 40	Muito Quente

Ex. 3.4 Defina uma função que receba uma idade inteira e decida se a pessoa é criança (0 a 12 anos), adolescente (13 a 17 anos), adulto (18 a 59 anos) ou idoso (60 anos ou mais). A classificação deve ser retornada como uma string. Se a idade for negativa, a função deve retornar uma string vazia.

Ex. 3.5 Defina uma função que recebe um número inteiro e retorne uma lista com 4 elementos `(m, c, d, u)`, onde:

- `m`: quantidade de milhares
- `c`: quantidade de centenas

- d: quantidade de dezenas
- u: quantidade de unidades

Por exemplo, se o número for 1234, a função deve retornar `List(1, 2, 3, 4)`. Se o número for 123456, a função deve retornar `List(123, 4, 5, 6)`. Se o número for negativo, a função deve retornar uma lista vazia.

Ex. 3.6 Um número é palíndromo quando possui a mesma sequência de dígitos tanto quando lido da esquerda para a direita quanto da direita para a esquerda. Por exemplo, 12321 é palíndromo, mas 45534 não é. Defina um predicado (isto é, uma função que retorna um valor booleano) que receba um inteiro de 5 dígitos e determine se o número é palíndromo. Se o número possuir mais de 5 dígitos, ou se for negativo, a função deve retornar `false`. Importante: Não é permitido converter o inteiro para string ou lista — use apenas operações aritméticas para resolver o problema.

2.12 Soluções dos exercícios

Ex. 3.1

```
def conceitoNota(n: Double): String = {
  if (n >= 0 && n <= 5.0) "F"
  else if (n > 5.0 && n < 7.0) "D"
  else if (n >= 7.0 && n < 9.0) "C"
  else if (n >= 9.0 && n < 10.0) "B"
  else if (n == 10.0) "A"
  else ""
}
```

Ex. 3.2

```
def comissao(qtd: Int): Double = {
  if (qtd < 0) 0.0
  else if (qtd > 500) qtd * 2.0
  else if (qtd > 250) qtd * 1.5
  else qtd * 1.0
}
```

Ex. 3.3

```
def classificaTemperatura(t: Double): List[String] = {
  if (t < -10.0) List("Congelante")
  else if (t >= -10.0 && t < 0.0) List("Muito Frio")
  else if (t == 0.0) List("Muito Frio", "Frio")
  else if (t < 10.0) List("Frio")
  else if (t == 10.0) List("Frio", "Fresco")
  else if (t < 20.0) List("Fresco")
```

```
  else if (t == 20.0) List("Fresco", "Agradável")
  else if (t < 30.0) List("Agradável")
  else if (t == 30.0) List("Agradável", "Quente")
  else if (t <= 40.0) List("Quente")
  else if (t > 40.0) List("Muito Quente")
  else List()
}
```

Ex. 3.4

```
def classificaIdade(idade: Int): String = {
  if (idade < 0) ""
  else if (idade <= 12) "criança"
  else if (idade <= 17) "adolescente"
  else if (idade <= 59) "adulto"
  else "idoso"
}
```

Ex. 3.5

```
def decompoeNumero(n: Int): List[Int] = {
  if (n < 0) List()
  else {
    val u = n % 10
    val d = (n / 10) % 10
    val c = (n / 100) % 10
    val m = n / 1000
    List(m, c, d, u)
  }
}
```

Ex. 3.6

```
def palindromoAritmetico(n: Int): Boolean = {
  if (n < 0 || n > 99999) false
  else {
    val d1 = (n / 10000) % 10
    val d2 = (n / 1000) % 10
    val d3 = (n / 100) % 10
    val d4 = (n / 10) % 10
    val d5 = n % 10
    (n >= 10000 && n <= 99999) &&
    (d1 == d5 && d2 == d4)
  }
}
```

Aviso de licença de uso