Capítulo 1

Introdução à programação funcional

A programação funcional é um paradigma de programação no qual os programas são construídos pela definição, aplicação e composição de funções puras. Para tal, programas funcionais evitam o uso de estados mutáveis e efeitos colaterais, o que os torna mais fáceis de entender, testar e manter. A programação funcional é um paradigma declarativo, onde o foco está na definição do que deve ser feito, em vez de como deve ser feito. As principais características da programação funcional são:

Funções de primeira classe e de ordem superior Funções podem ser tratadas como valores, podendo ser passadas como argumentos, retornadas de outras funções e atribuídas a variáveis.

Imutabilidade Dados são imutáveis por padrão, o que significa que uma vez criados, não podem ser alterados. Alterações são concretizadas pela criação de novas versões dos dados. Isso ajuda a evitar efeitos colaterais indesejados.

Funções puras São funções que não têm efeitos colaterais e sempre retornam o mesmo resultado para os mesmos argumentos.

Recursividade Os algoritmos que envolvem repetições são modelados como algoritmos recursivos.

Composição de funções Funções podem ser combinadas para criar novas funções, promovendo a reutilização de código.

1.1 Paradigmas de programação

Um paradigma de programação é um modo de classificar uma linguagem de programação de acordo com as suas características fundamentais. Um paradigma pode também ser compreendido como um estilo de programação. Como tal, podemos dizer que a programação funcional é um paradigma de programação, conhecido como o paradigma de programação funcional.

Ao longo do século XX, os estilos de programação evoluíram e foram tornando-se mais sofisticados. As primeiras abordagens eram consideradas de baixo nível, envolvendo diretamente a manipulação de memória e de registradores, seja utilizando linguagem de máquina (e.g. códigos binários) ou linguagem de montagem (e.g. linguagens Assembly[1]). Com o passar do tempo, surgiram abordagens mais sofisticadas, que permitiam a programação em um nível mais alto de abstração. A partir desse momento, podemos começar a falar em paradigmas de programação.

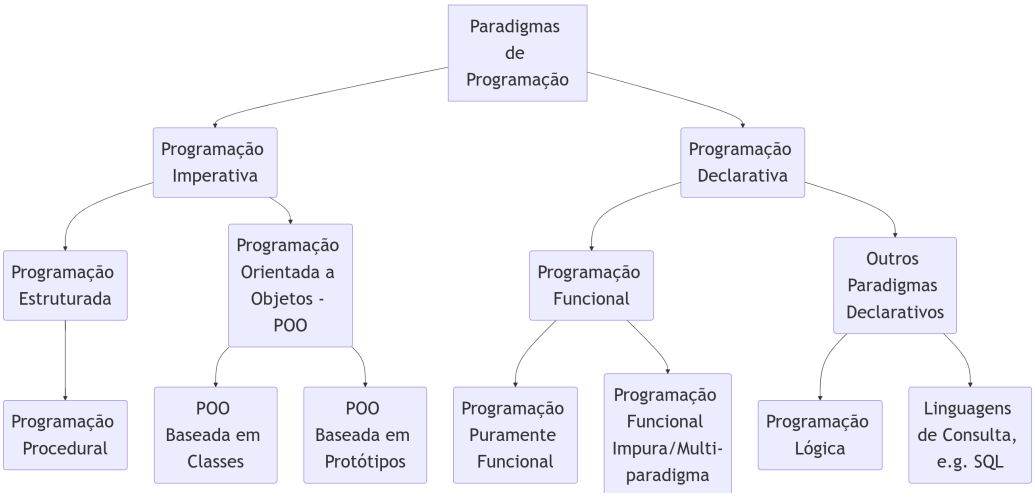

Figura 1.1: Diagrama simplificado dos principais paradigmas de programação

A figura 1.1 ilustra os principais paradigmas de programação e como eles se relacionam hierarquicamente. Embora seja uma simplificação, pois existem mais relacionamentos entre os paradigmas dos que os representados na figura, ela ajuda a entender como os paradigmas evoluíram ao longo do tempo e como eles se relacionam entre si. Alguns dos paradigmas mais importantes são:

[1] As linguagens Assembly são múltiplas, pois as instruções disponíveis dependem da arquitetura do processador ao qual são aplicadas (e.g. x86, ARM, MIPS, PowerPC, etc.)

Programação imperativa Descreve a computação em termos de estados mutáveis e comandos, executados sequencialmente, que alteram esses estados ao longo do tempo. É uma das formas clássicas de programação, pois remete fortemente a conceitos presentes nos modelos da máquina de Turing e da arquitetura de Von Neumann. A programação em linguagem de máquina ou em linguagem de montagem são exemplos de programação imperativa, mas classificamos também dentro desse paradigma as muitas linguagens de programação de mais alto nível criadas a partir dos anos 1960, como, por exemplo, Assembly, C, Pascal, Cobol, Fortran, Basic, etc.

Programação procedural Subconjunto da programação imperativa que descreve a computação em termos de procedimentos, que são blocos de código que podem ser chamados a partir de outros blocos de código. Comumente confundidos com funções, os procedimentos possuem uma definição mais abrangente, pois não precisam restringir-se à definição matemática de funções. Em suma, linguagens imperativas que fazem uso de procedimentos são classificadas como linguagens de programação procedurais. Exemplos de linguagens procedurais são: Assembly, C, Pascal, Fortran, Basic, etc.

Programação estruturada Trata-se de um subconjunto do paradigma imperativo, que introduz a noção de estruturas de controle de fluxo (sequência, seleção e repetição). A programação estruturada é um subconjunto mais organizado e legível que a programação imperativa e seus conceitos foram formalizados a partir dos anos 1960 por Edsger Dijkstra, Niklaus Wirth e Tony Hoare. A programação estruturada é a base para a programação imperativa moderna. Exemplos de linguagens estruturadas são: C, Pascal, Fortran, Basic, etc.

Programação orientada a objetos Evolução da programação estruturada que introduz o conceito de objetos, compostos por propriedades (variáveis de estado) e métodos (procedimentos que operam sobre o estado do objeto). A programação orientada a objetos é baseada nos conceitos de modularização, encapsulamento, passagem de mensagem, composição, herança e polimorfismo. Esses conceitos permitem a criação de programas mais organizados, reutilizáveis e fáceis de manter. Em grande parte das linguagens desse paradigma, objetos são instâncias de classes, embora em algumas linguagens objetos podem existir por si só (e.g. JavaScript). Exemplos de linguagens orientadas a objetos são: Java, C++, Python, Ruby, C#, etc.

Programação declarativa Descreve a computação em termos de lógica e restrições, sem explicitar o controle de fluxo. A programação declarativa é um paradigma de programação que descreve *o que* o programa deve realizar, e não *como* ele deve realizar. Trata-se de outro estilo clássico de programação, em geral fundamentado em diferentes lógicas matemáticas. Nesse paradigma estão incluídas as linguagens de consulta, como SQL[2], as linguagens de programação lógica, além das linguagens de programação funcional.

Programação funcional Subconjunto da programação declarativa que descreve a computação em termos de funções matemáticas e evita estados mutáveis. A programação funcional é fundamentada em cálculo lambda[3]. Como o próprio nome implica, todas as computações devem ser expressas em termos de variáveis imutáveis e funções recursivas. Nesse paradigma estão incluídas linguagens como Haskell, Clojure, Scala, Scheme, etc.

Programação lógica Subconjunto da programação declarativa que descreve a computação em termos de lógica formal. A depender da linguagem, diferentes lógicas podem ser usadas, por exemplo a lógica de primeira ordem, a lógica de segunda ordem, a lógica modal, etc. Características importantes da programação lógica, como o reconhecimento de padrões e a inferência lógica, tem sido incorporadas em linguagens de programação multiparadigmas (por exemplo, Erlang e Scala incorporam características da programação lógica). Exemplos de linguagens de programação lógica são: Prolog, Datalog, etc.

[2] Apesar de ser formalmente classificada como linguagem de consulta, SQL (*Structured Query Language*) também pode ser classificada como uma linguagem de programação declarativa pois versões modernas (a partir de 1999) são Turing-completas. O conceito de Turing-completude é um conceito fundamental da teoria da computabilidade e diz respeito à capacidade de um sistema computacional simular uma máquina de Turing. A linguagem SQL, com a adição *Common Table Expressions* (CTEs), permite a definição de consultas recursivas, o que a torna Turing-completa.

[3] O cálculo lambda é um sistema formal desenvolvido por Alonzo Church na década de 1930 para estudar a computabilidade e a lógica matemática. É a base teórica para a programação funcional. A tese de Church-Turing afirma que qualquer função computável pode ser computada por uma máquina de Turing. Em suma, podemos dizer que o cálculo lambda e a máquina de Turing são equivalentes em termos de computabilidade e, por consequência, qualquer programa imperativo pode ser traduzido para um programa funcional e vice-versa.

1.2 Tipos de linguagens funcionais

Embora o conceito de paradigma de programação tenha grande relevância para o estudo de linguagens de programação, na prática as linguagens de programação constumam ser multiparadigma. Por exemplo, a linguagem Scala é uma linguagem multiparadigma que promove a programação funcional como o paradigma de programação prioritário, mas também permite a programação orientada a objetos e a programação imperativa. Por outro lado, a linguagem JavaScript é uma linguagem multiparadigma que promove a programação funcional como um dos paradigmas de programação, mas não necessariamente como o paradigma de programação prioritário. Podemos classificar as linguagens de programação funcional nos seguintes:

Linguagens funcionais puras São aquelas que: i) aderem aos principios fundamentais da programação funcional, qual sejam: imutabilidade, ausência de efeitos colaterais e transparência referencial; ii) não permitem a definição de programas que violem quaisquer desses princípios. Exemplos de linguagens funcionais puras são Haskell, Curry, Elm, Mercury, etc.

Linguagens funcionais híbridas São aquelas que: i) aderem aos principios fundamentais da programação funcional; e ii) permitem a definição de programas que violem quaisquer desses princípios. De fato, as linguagens funcionais híbridas são linguagens multiparadigma que promovem o paradigma funcional em diferentes níveis de prioridade.

Dentro desse grupo, é oportuno distinguir duas subcategorias:

Linguagens funcionais híbridas com foco funcional São aquelas que promovem o paradigma funcional como o paradigma de programação prioritário, mas permitem o uso de outros paradigmas, como o imperativo, o procedural e o orientado a objetos. Também conhecidas como *functional-first*. Exemplos de linguagens funcionais híbridas com foco funcional são: Scala, Clojure, F#, OCaml, etc.

Linguagens funcionais híbridas com foco imperativo São aquelas que promovem o paradigma funcional como um dos paradigmas de programação, mas não necessariamente como prioritário. Também conhecidas como *imperative-first* ou *oo-first* (remete a orientação a objetos). Essas linguagens permitem o uso do paradigma funcional, mas promovem outros paradigmas como prioritários, a exemplo do

imperativo, do procedural e do orientado a objetos. Exemplos de linguagens funcionais híbridas com foco imperativo são: JavaScript, Python, Ruby, Java, Kotlin, etc.

Como podemos observar, a programação funcional é um paradigma de programação que pode ser aplicado em diferentes linguagens de programação, sejam elas funcionais puras ou híbridas. A escolha da linguagem de programação funcional depende do contexto e dos requisitos do projeto, bem como das preferências e habilidades dos desenvolvedores.

1.3 Histórico da programação funcional

A programação funcional desenvolveu-se em paralelo com a programação imperativa, mas com um foco diferente. Do ponto de vista teórico, a programação imperativa apoia-se na teoria de computabilidade estabelecida por Alan Turing, que introduziu o conceito de máquina de Turing. Já a programação funcional tem suas raízes na teoria de computabilidade formalizada por Alonzo Church, com o cálculo lambda. Posteriormente, demonstrou-se que ambos os modelos de computação são equivalentes, ou seja, qualquer função computável pode ser expressa tanto em termos de máquina de Turing quanto em termos de cálculo lambda. Essa equivalência é conhecida como a tese de Church-Turing. A seguir, detalhamos os principais marcos históricos da programação funcional.

- **1930s**. Alonzo Church desenvolve o cálculo lambda, um sistema formal para expressar funções matemáticas e computação. O cálculo lambda é a base teórica da programação funcional. Tanto Church como Turing desenvolveram suas teses em meados de 1936, sendo a equivalência entre os modelos efetuada pelo próprio Turing e por outros pesquisadores posteriormente.

- **1950s.** A programação funcional começa a ganhar popularidade com o desenvolvimento de linguagens como Lisp (por John McCarthy) e suas derivadas, que foram projetadas para suportar a programação funcional de forma mais eficiente. O desenvolvimento de Lisp coincide também com o desenvolvimento da inteligência artificial, que utilizava a programação funcional como uma das suas principais abordagens.

- **1960s.** O conceito de programação funcional é formalizado por Peter Landin, que introduz o conceito de *expressões* e *valores* em programação. Lan-

din também introduz o conceito de *continuations*, que são uma forma de controlar o fluxo de execução em programas funcionais.

- **1970s.** A programação funcional continua a evoluir com o desenvolvimento de linguagens como ML, que introduzir inferência de tipos e casamento de padrões, características importantes da programação funcional atualmente. Em 1977, John Backus, um dos criadores de Fortran, defende a programação funcional durante seu discurso, denominado "Can Programming Be Liberated from the von Neumann Style?", ocasião em que recebeu o prêmio Turing.

- **1980s.** A programação funcional começa a ser adotada numa escala maior, com o desenvolvimento de linguagens como Miranda, uma das principais linguagens funcionais puras, com avaliação preguiçosa. Em 1986, é criada Erlang, uma linguagem funcional híbrida com foco em concorrência e sistemas distribuídos, que se tornou popular no setor de telecomunicações a partir da década de 1990.

- **1990s.** A programação funcional continua a evoluir e a ganhar popularidade com o surgimento de novas linguagens e paradigmas. O desenvolvimento de linguagens como Haskell, que se tornou uma das principais linguagens funcionais puras, e a adoção de conceitos funcionais em linguagens imperativas utilizadas para desenvolvimento Web, como JavaScript, Python e Ruby, que incorporaram características da programação funcional, como funções de primeira classe, funções de ordem superior e imutabilidade.

- **2000s.** O crescimento exponencial dos dados na Web gera o movimento de *Big Data*, que por sua vez impulsiona a adoção de programação funcional, com o uso de linguagens funcionais e abstrações do paradigma funcional para processamento de dados em larga escala em ambientes de *Cloud Computing*. Desenvolvem-se novas linguagens, como Clojure, Scala e F#, que incorporam características *functional-first* e permitem a programação concorrente de forma mais produtiva.

- **2010s.** A ascensão da computação móvel consolida ainda mais os grandes volumes de dados como motor do setor de tecnologia, criando profissões como *Data Scientist* e *Data Engineer*, que tendem a adotar o paradigma funcional para manipulação e análise de dados. As linguagens imperativas majoritárias incorporam características de programação funcional. Java, C++ e

C# introduzem suporte a programação funcional, como expressões lambda, funções de ordem superior e imutabilidade controlada. Isso torna a programação funcional mais acessível para desenvolvedores que trabalham com essas linguagens.

O uso de GPUs (unidades de processamento gráfico) para computação paralela para treinar redes neurais profundas e algoritmos de aprendizado de máquina também impulsiona a adoção de programação funcional em vertentes mais híbridas. Além disso, o movimento *Functional Programming in JavaScript* ganha força, com o uso de bibliotecas como Ramda, Lodash, React, Redux e RxJS para facilitar a programação funcional em JavaScript, com foco prioritário no front-end de sistemas web. Posteriormente, movimento similar propaga para a computação móvel, com o uso de bibliotecas como Jetpack Compose, React Native, Flutter, etc.

- **2020s.** O paradigma funcional se consolida no desenvolvimento de software comercial, com o uso de linguagens funcionais puras e híbridas em diversas áreas, como ciência de dados, engenharia de dados, inteligência artificial, sistemas distribuídos e programação concorrente.

  Evoluções no modelo de serviços computacionais, como serverless computing, Function as a Service (FaaS), Event-Driven Architecture (EDA), Event Sourcing, arquiteturas reativas, conteinização e microserviços, impulsionam a adoção de conceitos relacionais à programação funcional, como imutabilidade, transparência referencial e componibilidade de funções. Essas abordagens permitem a criação de sistemas mais escaláveis, resilientes e fáceis de manter, aproveitando as vantagens da programação funcional.

1.4 Fundamentos da programação funcional

A programação funcional é baseada em alguns princípios fundamentais que a diferenciam de outros paradigmas de programação. Esses princípios são:

- Funções de primeira classe
- Funções de ordem superior
- Imutabilidade
- Ausência de efeitos colaterais

- Funções puras
- Transparência referencial
- Recursividade

Vamos explorar cada um desses princípios em mais detalhes.

1.4.1 Funções de primeira classe

Definição Uma linguagem de programação possui funções de primeira classe quando as funções podem ser tratadas como valores, ou seja, as funções podem ser:

- Atribuídas a variáveis;
- Passadas como argumentos para outras funções;
- Retornadas como valores de outras funções;
- Armazenadas em estruturas de dados, como listas, conjuntos, dicionários, etc.

Lambdas Em linguagens funcionais, as funções são frequentemente representadas como *funções anônimas*, conhecidas também como *lambdas* — o nome deriva da letra grega usada por Alonzo Church para representar funções no cálculo lambda. As lambdas são funções que não possuem um nome e são definidas de forma concisa, geralmente em uma única linha de código. As lambdas possuem os seguintes componentes:

- **Cabeça**: a função lambda pode receber parâmetros de entrada, que são os valores que serão utilizados dentro da função.
- **Corpo**: o corpo da função lambda é o código que será executado quando a função for chamada. O corpo pode conter uma ou mais expressões, que serão avaliadas quando a função for chamada.

Linguagem	Exemplo de lambda
JavaScript	`(x) => x + 1`
Python	`lambda x: x + 1`
Ruby	`|x| x + 1`
Java	`(x) -> x + 1`
Scala	`x => x + 1`
Haskell	`\x -> x + 1`

Tabela 1.1: Exemplos de lambdas em diferentes linguagens de programação

Na maioria das linguagens, a cabeça e o corpo da lambda são separados por um símbolo especial, como a seta (`->`) ou o símbolo de dois pontos (`:`), como podemos observar na tabela 1.1 ilustra como as lambdas são representadas em diferentes linguagens de programação.

Representação de funções Em linguagens multiparadigma, como JavaScript, Python e Ruby, as funções são representadas como objetos. Isso facilita bastante a representação, pois objetos são elementos de primeira classe nessa linguagem. Essa decisão de projeto evidencia a possibilidade de combinar os paradigmas de programação funcional e orientada a objetos, permitindo que as funções sejam tratadas como valores e possam ser manipuladas como tal.

Funções de primeira classe em Java A partir da versão 8.0 (2014) de Java, a linguagem passou a suportar funções de primeira classe, por meio de extensões à linguagem para permitir o uso de lambdas. A implementação desse recurso considerou a retrocompatibilidade com versões anteriores da linguagem, de modo que lambdas também são representadas como objetos. Mais particular ainda é a restrição de que esses objetos "especiais" devem ser necessariamente tipados com um tipo específico de interface, denominada interface funcional.

Interfaces funcionais A implementação de lambdas em Java é feita por meio de *interfaces funcionais*, que são interfaces que devem possuir apenas um método abstrato, ou seja, deve ser uma interface SAM (*Single Abstract Method*). As lambdas são representadas como instâncias de classes anônimas que implementam essas interfaces funcionais, permitindo que sejam tratadas como valores de primeira classe. Um aspecto interessante é que o programador não precisa criar as classes anônimas, pois o compilador Java as gera automaticamente a partir da definição da lambda. Considere o programa 1.1, no qual são definidas algumas

interfaces funcionais.

Listagem 1.1: Exemplos de interface funcional em Java

```
1  @FunctionalInterface
2  private static interface UnaryFunction<T, R> {
3     R apply(T t);
4  }
5
6  @FunctionalInterface
7  private static interface BinaryFunction<T, U, R> {
8     R apply(T t, U u);
9  }
10
11 @FunctionalInterface
12 private static interface TernaryFunction<T, U, V, R> {
13    R apply(T t, U u, V v);
14 }
```

No programa, são definidas três interfaces funcionais: `UnaryFunction`, `BinaryFunction` e `TernaryFunction`. Essas interfaces representam funções que recebem um, dois e três argumentos, respectivamente, e retornam um valor. Formalmente, estamos definindo tipos de função de acordo com a aridade (quantidade de argumentos) da função. Além disso, usamos tipos genéricos para permitir que as interfaces funcionais sejam reutilizadas com diferentes tipos de dados. Isso torna a interface funcional aplicável a um conjunto maior de possíveis lambdas. Por exemplo, a interface `BinaryFunction` pode ser utilizada para definir qualquer função que receba dois argumentos e retorne um valor, independentemente dos tipos específicos utilizados.

A anotação `@FunctionalInterface` A anotação `@FunctionalInterface` é uma marcação que indica que a interface é uma interface funcional. Essa anotação é opcional, mas é recomendada, pois permite que o compilador verifique se a interface realmente possui apenas um método abstrato. Se a interface tiver mais de um método abstrato, o compilador irá gerar um erro. É imprescindível que uma interface funcional possua apenas um método abstrato, pois caso contrário haveria ambiguidade sobre qual método chamar para aplicar a lambda.

Definição de lambdas Toda lambda precisa ser tipada com uma interface funcional, que define a assinatura da função. Na listagem 1.2, são definidas algu-

mas lambdas que implementam as interfaces funcionais definidas anteriormente.

Listagem 1.2: Exemplos de definição de lambdas em Java

```
1 UnaryFunction<Integer, Integer> square = x -> x * 2;
2
3 BinaryFunction<Integer, Integer, Integer> add = (x, y) -> x + y;
4 BinaryFunction<Integer, Integer, Integer> multiply = (x, y) -> x
     * y;
5
6 TernaryFunction<Integer, Integer, Integer, Integer> sumThree =
     (x, y, z) -> x + y + z;
```

Nesses exemplos podemos observar a possibilidade armazenar lambdas em variáveis, que são tipadas com as interfaces funcionais definidas anteriormente. As lambdas são definidas de forma concisa, utilizando a sintaxe de seta (`->`) para separar os parâmetros do corpo da função. Caso a lambda não corresponda à assinatura da interface funcional, o compilador irá gerar um erro de tipo. Por exemplo, listagem 1.3 apresenta um erro de tipo, pois a lambda definida não corresponde à assinatura da interface funcional `UnaryFunction`, que espera apenas um argumento, sendo que a lambda definida recebe dois argumentos.

Listagem 1.3: Exemplo de erro de tipo em Java

```
1 UnaryFunction<Integer, Integer> square = (x, y) -> x * y;
```

```
FirstClassFunctions.java:48: error: incompatible types:
   incompatible parameter types in lambda expression
     UnaryFunction<Integer, Integer> square2 = (x, y) -> x * y;
```

Além disso, não é possível utilizar inferência de tipos, via o modificador `var`, para definir lambdas, pois a sintaxe em Java não permite tipar explicitamente os parâmetros nem o retorno da função. Com isso, o compilador não consegue inferir o tipo da lambda a partir do contexto. Por exemplo, na listagem 1.4, o compilador não consegue inferir o tipo da lambda, pois a variável `square` não é tipada com uma interface funcional.

Listagem 1.4: Exemplo de erro de inferência de tipos em Java

```
1 var square = x -> x * 2; // Erro de inferência de tipos
```

```
FirstClassFunctions.java:48: error: cannot infer type for local
   variable square2
      var square2 = (x, y) -> x * y;
 (lambda expression needs an explicit target-type)
```

Aplicação de lambdas Em programação funcional, a operação de aplicação de uma função é o processo de executar a função com os argumentos fornecidos. Em Java, a aplicação de uma lambda é feita chamando o único método abstrato da interface funcional que a define. Por exemplo, na listagem 1.5, aplicamos as lambdas definidas anteriormente.

Listagem 1.5: Exemplo de aplicação de lambdas em Java

```
1 System.out.println("Square of 5: " + square.apply(5));
2
3 System.out.println("Sum of 3 and 4: " + add.apply(3, 4));
4 System.out.println("Product of 3 and 4: " + multiply.apply(3,
     4));
5
6 System.out.println("Sum of 1, 2, and 3: " + sumThree.apply(1, 2,
     3));
```

```
> javac FunctionalInterfaceDemo.java
> java FunctionalInterfaceDemo
Square of 5: 10
Sum of 3 and 4: 7
Product of 3 and 4: 12
Sum of 1, 2, and 3: 6
```

Interfaces funcionais pré-definidas A biblioteca-padrão de Java possui diversas interfaces funcionais pré-definidas, que podem ser utilizadas para definir lambdas. Sempre que possível, é recomendável utilizar essas interfaces pré-definidas, pois elas são amplamente utilizadas e bem documentadas, o que aumenta a compatibilidade do código com APIs e bibliotecas existentes. Essas interfaces estão localizadas no pacote `java.util.function` e incluem interfaces como `Function`, `Consumer`, `Supplier`, `Predicate`, entre outras. Por exemplo, poderíamos modificar parcialmente o programa da listagem 1.2 para utilizar a interface

funcional `Function` do pacote `java.util.function`, ao invés das nossas interfaces funcionais customizadas, como na listagem 1.6.

Listagem 1.6: Exemplo de definição de lambdas utilizando a interface Function em Java

```
1 Function<Integer, Integer> square = x -> x * x;
2
3 BiFunction<Integer, Integer, Integer> add = (x, y) -> x + y;
4 BiFunction<Integer, Integer, Integer> multiply = (x, y) -> x * y;
```

Na listagem 1.6, utilizamos a interface funcional `Function` e `BiFunction` para definir funções de aridade 1 e 2, de modo bastante similar ao que fizemos com as interfaces `UnaryFunction` e `BinaryFunction`, respectivamente.

Uma limitação importante do pacote `functions` é que ele não prevê, naturalmente, todas as aridades de funções, ou seja, não existem interfaces funcionais para funções de aridade maior que 2. Nesses casos, torna-se necessário definir interfaces funcionais customizadas, como a interface `TernaryFunction` definida anteriormente.

1.4.2 Funções de ordem superior

Definição Uma vez que a linguagem de programação possui funções de primeira classe, podemos definir as *funções de ordem superior*, que são funções que:

- Recebem outras funções como argumentos; e/ou
- Retornam outras funções como resultado.

Funções como argumentos As funções de ordem superior são amplamente utilizadas em programação funcional para criar abstrações e compor funções. Por exemplo, podemos definir uma função de ordem superior que recebe uma função como argumento e a aplica a uma lista de valores, como na listagem 1.7.

Listagem 1.7: Exemplo de função de ordem superior em Java

```
1 public static ArrayList<Integer> map(ArrayList<Integer> arr,
    Function<Integer, Integer> func) {
```

```
    var result = new ArrayList<Integer>(arr.size());

    arr.forEach(item -> result.add(func.apply(item)));

    return result;
}
```

Por questões de simplicidade, optamos por representar um método estático que recebe uma lista de inteiros e uma função de ordem superior, que será aplicada a cada elemento da lista. Conceitualmente, um método pode ser considerado como uma função, portanto o conceito de ordem superior pode ser aplicado a métodos também, sejam eles estáticos ou de instância. A partir desse ponto, vamos utilizar os termos métodos e funções de forma intercambiável, exceto quando for necessário distinguir entre os dois conceitos.

A função `map` retorna uma nova lista com os resultados da aplicação da função a cada elemento da lista original. De fato, essa função de ordem superior implementa um padrão de projeto clássico da programação funcional, conhecido como *map*, que é utilizado para transformar uma coleção de valores aplicando uma função a cada elemento da coleção. A listagem 1.8 ilustra como podemos utilizar a função `map` para transformar uma lista de inteiros, aplicando diferentes lambdas para calcular o quadrado e o dobro de cada elemento da lista.

Listagem 1.8: Exemplo de uso de função de ordem superior em Java

```
ArrayList<Integer> squares = map(numbers, x -> x * x);
System.out.println("Squares: " + squares);
ArrayList<Integer> doubles = map(numbers, x -> x * 2);
System.out.println("Doubles: " + doubles);
```

```
Squares: [1, 4, 9, 16, 25]
Doubles: [2, 4, 6, 8, 10]
```

Funções como retorno As funções de ordem superior também podem retornar outras funções como resultado. Por exemplo, podemos definir um método que retorna uma função que realiza uma operação matemática específica, como adição, subtração, multiplicação ou divisão, conforme a listagem 1.9. Essa função de ordem superior pode ser utilizada para criar funções específicas para cada operação matemática, atuando como uma fábrica de funções.

Listagem 1.9: Exemplo de função de ordem superior que retorna outra função em Java

```
public static BiFunction<Integer, Integer, Integer>
    genOperation(Operation op) {
   switch (op) {
      case ADD:
         return (a, b) -> a + b;
      case SUBTRACT:
         return (a, b) -> a - b;
      case MULTIPLY:
         return (a, b) -> a * b;
      case DIVIDE:
         return (a, b) -> {
            if (b == 0) {
               throw new ArithmeticException("Division by zero");
            }
            return a / b;
         };
   }
   throw new IllegalArgumentException("Invalid operation");
}
```

A listagem 1.9 define uma função de ordem superior chamada `genOperation`, que recebe um valor do tipo `Operation` e retorna uma função que realiza a operação matemática correspondente. A enumeração `Operation` é definida na listagem 1.10.

Listagem 1.10: Exemplo de enumeração para operações matemáticas em Java

```
public enum Operation {
   ADD,
   SUBTRACT,
   MULTIPLY,
   DIVIDE
}
```

A listagem 1.11 ilustra como podemos utilizar a função `genOperation` para criar funções específicas para cada operação matemática e aplicá-las a dois números.

Listagem 1.11: Exemplo de uso de função de ordem superior que retorna outra função em Java

```
1 for (Operation op : Operation.values()) {
2    BiFunction<Integer, Integer, Integer> operation =
        genOperation(op);
3    System.out.println("Operation: " + op);
4    System.out.println("Result of 10 and 5: " +
        operation.apply(10, 5));
5 }
```

```
Operation: ADD
Result of 10 and 5: 15
Operation: SUBTRACT
Result of 10 and 5: 5
Operation: MULTIPLY
Result of 10 and 5: 50
Operation: DIVIDE
Result of 10 and 5: 2
```

1.4.3 Imutabilidade

A imutabilidade é um dos princípios mais importantes da programação funcional. Consiste na impossibilidade de alterar o estado de um dado após a sua criação. Uma vez que um dado seja inicializado, ele não pode ser modificado durante todo o seu tempo de vida. Na prática, isso traz algumas implicações para os componentes de um programa:

- **Estado constante**. Se um dado é imutável, então ele possui apenas um estado ao longo do seu tempo de vida.
- **Persistência.** Sempre que precisamos alterar um dado, devemos criar uma nova versão desse dado, com o novo estado. Isso significa que mantemos um histórico de versões dos dados, tornando as alterações não destrutivas.

O PROBLEMA DA MUTABILIDADE A mutabilidade é uma característica pervasiva na programação imperativa, pois facilita a manipulação de dados e o controle

de fluxo. De fato, quando recuperamos o fundamento teórico da programação imperativa, a máquina de Turing, a mutabilidade é inerente ao modelo, que remete à escrita na fita da máquina. Essa característica traz algumas consequências:

- **Destrutividade.** A mutabilidade traz como característica intrínseca a destrutividade, ou seja, ao alterar o estado de um dado, perdemos o estado anterior de modo irreversível.
- **Imprevisibilidade.** A imutabilidade torna os programas altamente dependente da ordenação das intruções. Em programas que possuem mais de uma linha de execução, como os programas concorrentes, essas ordenações podem se intercalar, podendo levar a resultados imprevisíveis, pois diferentes linhas de execução podem alterar o estado dos dados ao mesmo tempo. Isso pode levar a condições de corrida e outros problemas de concorrência.
- **Efeitos colaterais.** A mutabilidade pode levar a efeitos colaterais indesejados, pois a alteração do estado de um dado pode propagar efeitos para outras partes do programa que dependem desse dado. Isso torna o código mais difícil de entender e manter.

Contenção da mutabilidade Um dos piores cenários para lidar com a mutabilidade consiste em um programa em que todas as variáveis são globais e mutáveis. Nesse cenário, qualquer parte do código pode alterar o estado de qualquer variável, tornando o programa difícil de entender, de testar e de manter.

Para lidar com a mutabilidade, a programação imperativa desenvolveu diversas técnicas para "conter" os efeitos colaterais da mutabilidade. Essas técnicas buscam organizar o código de forma que os efeitos colaterais sejam previsíveis e controlados, tornando o código mais legível e fácil de entender. Grande parte dos desenvolvimentos em programação imperativa, como por exemplo, a programação estruturada, a programação procedural e a programação orientada a objetos, foram, em grande parte, motivados pela necessidade de lidar com a mutabilidade de forma mais organizada e previsível. Algumas dessas técnicas são:

- **Modularização e abstração.** A modularização é uma técnica que busca dividir o código em módulos, que são unidades independentes de código que podem ser reutilizadas e testadas separadamente. Isso ajuda a conter os efeitos colaterais da mutabilidade, pois cada módulo pode ter seu próprio estado e suas próprias regras de mutabilidade. Relacionado ao conceito de

abstração, a modularização permite que o projetista do módulo defina uma interface pública para o módulo, escondendo os detalhes de implementação e minimizando os efeitos colaterais indesejados.

- **Ocultamento de informação.** O ocultamento de informação é um princípio que norteia o projeto de módulos, prescrevendo que detalhes de sua implementação sejam ocultados, expondo apenas uma interface pública. Isso ajuda a conter os efeitos colaterais da mutabilidade, pois o estado do módulo é acessado apenas por meio de métodos públicos, que são desenvolvidos pelo projetista do módulo.

  O **encapsulamento** é uma técnica concreta para obter ocultamento de informação, comumente utilizado nos tipos abstratos de dados, de modo geral, e em classes e objetos, na programação orientada a objetos, em específico. Consiste em agrupar dados e operações em um único módulo, que se responsabiliza pela manutenção dos estados desses dados.

  O **controle de acesso** é uma técnica que busca restringir o acesso a determinados dados e métodos de um módulo, permitindo que apenas partes autorizadas do código possam acessá-los (e.g. modificadores de acesso em linguagens orientadas a objeto, diretivas de exportação em módulos, etc.)

- **Escopos de variáveis.** Na programação estruturada, o código pode ser organizado em estruturas de controle, cada um com seu próprio escopo. Isso permite que variáveis e estados tenham seu tempo de vida e visibilidade limitados a um trecho do código, minimizando os efeitos colaterais da mutabilidade.

  Na programação procedural e, por conseguinte, na orientação a objetos, escopos são também definidos nos corpos dos procedimentos, métodos, objetos ou classes. Um escopo fundamentalmente controla a visibilidade dos dados que contém e, como tal, é uma manifestação concreta do princípio de ocultamento de informação.

- **Imutabilidade controlada.** A imutabilidade controlada é uma técnica que busca tornar algumas partes do código imutáveis, enquanto outras partes permanecem mutáveis. Isso ajuda a conter os efeitos colaterais da mutabilidade, pois as partes imutáveis do código são mais fáceis de entender e manter.

- **Passagem de mensagens**. A passagem de mensagens é uma técnica que busca comunicar entre módulos por meio de mensagens, em vez de compartilhar estado. Isso ajuda a conter os efeitos colaterais da mutabilidade, pois cada módulo pode ter seu próprio estado e suas próprias regras de mutabilidade, sem interferir no estado dos outros módulos.
- **Composição.** A composição é uma técnica que busca combinar módulos para criar novos módulos, promovendo a reutilização de código. Isso ajuda a conter os efeitos colaterais da mutabilidade, pois cada módulo pode ter seu próprio estado e suas próprias regras de mutabilidade, sem interferir no estado dos outros módulos.

Mutabilidade e concorrência Como já mencionamos, a mutabilidade é uma característica pervasiva na programação imperativa, que pode levar a efeitos colaterais indesejados. Um exemplo clássico é a presença de variáveis mutáveis como estado compartilhado entre diferentes threads de execução, o que pode levar a condições de corrida e outros problemas de concorrência. Por exemplo, considere o programa 1.12, em Java.

Listagem 1.12: Exemplo de condição de corrida

```
1  public class ConcurrentSumDemo {
2     private static int sum = 0;
3
4     public static void main(String[] args) throws
          InterruptedException {
5        final int max = 10_000;
6        final int mid = max / 2;
7
8        Runnable sumFirstHalf = new Runnable() {
9           @Override
10          public void run() {
11             for (int i = 1; i <= mid; i++) {
12                sum += i;
13             }
14          }
15       };
16
17       Runnable sumSecondHalf = new Runnable() {
```

```
18          @Override
19          public void run() {
20             for (int i = mid + 1; i <= max; i++) {
21                sum += i;
22             }
23          }
24       };
25
26       Thread t1 = new Thread(sumFirstHalf);
27       Thread t2 = new Thread(sumSecondHalf);
28
29       t1.start();
30       t2.start();
31
32       t1.join();
33       t2.join();
34
35       System.out.println("Final sum: " + sum);
36    }
37 }
```

```
> javac ConcurrentSumDemo.java
> java ConcurrentSumDemo
Final sum: 44859591
> java ConcurrentSumDemo
Final sum: 50005000
> java ConcurrentSumDemo
Final sum: 48956560
> java ConcurrentSumDemo
Final sum: 46532078
```

O objetivo do programa é calcular a série $1 + 2 + \ldots + 10000$, que é igual a 50005000. No entanto, o programa apresenta uma condição de corrida, pois as duas threads estão acessando e modificando a variável compartilhada `counter` ao mesmo tempo. Isso pode levar a resultados imprevisíveis, como podemos observar na saída do programa. O resultado final do programa pode variar a cada execução, pois depende da ordem em que as threads são executadas. Isso torna o programa imprevisível e difícil de depurar. Existem diferentes soluções para esse problema,

como veremos a seguir.

Mutabilidade e concorrência imperativa Podemos usar técnicas de controle de concorrência explícito, como o uso de *locks* (bloqueios), sincronização, semáforos, variáveis atômicas, etc., para garantir que apenas uma thread possa acessar a variável compartilhada por vez. Java, por ser uma linguagem que também opera no paradigma concorrente, possui sintaxe própria para definir um monitor de modo fácil e intuitivo, como o uso do modificador `synchronized`. Veja por exemplo o programa 1.13, que usa o modificador `synchronized` para garantir que apenas uma thread possa acessar a variável `sum` por vez.

Listagem 1.13: Exemplo de condição de corrida com sincronização

```
1  public class ConcurrentSumImperativeDemo {
2     private static int sum = 0;
3     private static final Object lock = new Object();
4
5     public static void main(String[] args) throws
          InterruptedException {
6        final int max = 10_000;
7        final int mid = max / 2;
8        sum = 0;
9
10       Runnable sumFirstHalf = new Runnable() {
11          @Override
12          public void run() {
13             for (int i = 1; i <= mid; i++) {
14                synchronized (lock) {
15                   sum += i;
16                }
17             }
18          }
19       };
20
21       Runnable sumSecondHalf = new Runnable() {
22          @Override
23          public void run() {
24             for (int i = mid + 1; i <= max; i++) {
25                synchronized (lock) {
```

```
26                      sum += i;
27                  }
28              }
29          }
30      };
31
32      Thread t1 = new Thread(sumFirstHalf);
33      Thread t2 = new Thread(sumSecondHalf);
34
35      t1.start();
36      t2.start();
37
38      t1.join();
39      t2.join();
40
41      System.out.println("Final sum: " + sum);
42  }
43 }
```

```
> javac ConcurrentSumImperativeDemo.java
> java ConcurrentSumImperativeDemo
Final sum: 50005000
> java ConcurrentSumImperativeDemo
Final sum: 50005000
> java ConcurrentSumImperativeDemo
Final sum: 50005000
> java ConcurrentSumImperativeDemo
Final sum: 50005000
```

O programa não apresenta mais condições de corrida, pois o uso do modificador `synchronized` garante que apenas uma thread possa acessar a variável `sum` por vez. No entanto, essa abordagem pode levar a problemas de desempenho, como a espera ociosa, à medida que aumentamos o número de threads concorrentes, pois cada thread precisa esperar pela liberação do bloqueio para acessar a variável compartilhada. Além disso, o uso de *locks* e sincronização pode levar a problemas de complexidade no código, tornando-o mais difícil de entender e manter.

Imutabilidade e concorrência declarativa Outra abordagem é usar

técnicas de concorrência declarativa, que buscam evitar o uso de estados mutáveis compartilhados. Essas técnicas utilizam estados imutáveis, combinadas com o uso de abstrações como *futures* e *promises*, permitindo a composição de operações assíncronas sem a necessidade de estados mutáveis compartilhados. Isso tende a tornar o código mais legível, declarativo, fácil de entender e menos propenso a erros. Considere o programa 1.14, que usa a classe `Future` para calcular a soma de forma concorrente, sem o uso de estados mutáveis compartilhados.

Listagem 1.14: Exemplo de soma concorrente com *futures*

```
1  import java.util.concurrent.Callable;
2  import java.util.concurrent.ExecutionException;
3  import java.util.concurrent.FutureTask;
4  import java.util.stream.IntStream;
5
6  public class ConcurrentSumDeclarativeDemo {
7     public static void main(String[] args) throws
          InterruptedException, ExecutionException {
8        final int max = 10_000;
9        final int mid = max / 2;
10
11       Callable<Integer> sumFirstHalf = () ->
             IntStream.rangeClosed(1, mid).sum();
12       Callable<Integer> sumSecondHalf = () ->
             IntStream.rangeClosed(mid + 1, max).sum();
13
14       FutureTask<Integer> future1 = new
             FutureTask<>(sumFirstHalf);
15       FutureTask<Integer> future2 = new
             FutureTask<>(sumSecondHalf);
16
17       Thread t1 = new Thread(future1);
18       Thread t2 = new Thread(future2);
19
20       t1.start();
21       t2.start();
22
23       int result = future1.get() + future2.get();
24       System.out.println("Final sum: " + result);
```

```
25     }
26 }
```

```
> javac ConcurrentSumDeclarativeDemo.java
> java ConcurrentSumDeclarativeDemo
Final sum: 50005000
> java ConcurrentSumDeclarativeDemo
Final sum: 50005000
> java ConcurrentSumDeclarativeDemo
Final sum: 50005000
> java ConcurrentSumDeclarativeDemo
Final sum: 50005000
```

O programa não apresenta mais condições de corrida, pois o uso de *futures* permite que cada thread calcule sua parte da soma de forma independente, sem a necessidade de estados mutáveis compartilhados. Além disso, o uso de *futures* permite que o código seja mais declarativo e fácil de entender, pois expressa claramente a intenção de calcular a soma de forma concorrente.

Uma vantagem inerente da abordagem declarativa é que evita o uso de estados mutáveis compartilhados, o que torna o código mais legível e fácil de entender. Além disso, a abordagem declarativa permite a composição de operações assíncronas, o que pode levar a um melhor desempenho em programas concorrentes. No entanto, essa abordagem pode exigir um maior entendimento dos conceitos de concorrência e programação assíncrona, o que pode ser um desafio para desenvolvedores menos experientes.

1.5 Efeitos colaterais

O conceito de efeito colateral está fortemente relacionado ao conceito de expressões em linguagens de programação. Antes de adentrarmos na discussão sobre efeitos colaterais, é oportuno definir o que é uma expressão e o que é um efeito colateral.

Definição de expressão Uma expressão é uma combinação de valores, variáveis, operadores e funções que é avaliada para produzir um valor. Em outras palavras, uma expressão é qualquer trecho de código que pode ser avaliado para produzir um resultado. Por exemplo, as seguintes linhas de código são expressões em Java:

- `1 + 2` é uma expressão que produz o valor `3`.
- `Math.sqrt(4) * 3` é uma expressão que produz o valor `6.0`.
- `f(g(3) + h(4))` é uma expressão que produz um valor dependente das implementações das funções `f`, `g` e `h`.
- etc.

Em oposição às expressões, linguagens de programação também possuem *comandos* (*statements*), que são trechos de código que não produzem um valor, mas sim realizam uma ação. Por exemplo, as seguintes linhas de código são comandos em Java:

- `if (x > 0) { System.out.println("x é positivo"); }`, é um comando que executa uma ação (imprimir uma mensagem) se a condição for verdadeira, mas não produz um valor.
- `for (int i = 0; i < 10; i++) { x += i; }`, é um comando que executa uma ação (acumular os valores do contador) em um loop, mas não produz um valor.
- `int x = 5` é um comando que declara uma variável, mas não produz um valor.
- `System.exit(0)` é um comando que encerra o programa, mas não produz um valor.

Definição de efeito colateral Em programaçao, um efeito colateral refere-se a qualquer mudança observável ou interação com o ambiente externo efetuado por uma expressão, que não seja relacionado à resolução do seu valor. Em linhas gerais, qualquer operação que faça algo além de calcular um valor de resultado pode ser considerado um efeito colateral. Exemplos clássicos de efeitos colaterais incluem:

- **Acesso e modificação de dados mutáveis externos**. Uma função, por exemplo, gera efeito colateral se acessa ou modifica dados mutáveis que estão fora do seu escopo ou contexto. Por exemplo:
  - Acessar ou modificar uma variável global ou de classe.

  - Acessar ou modificar o estado interno de um objeto passado como argumento à função.
  - Modificar uma estrutura de dados mutável, como uma lista ou um dicionário, que foi instanciada fora do escopo da função.
  - etc.

- **Realização de operações de entrada/saída (I/O)**. Uma função gera efeito colateral sempre que realiza operações de entrada/saída. Por exemplo:
  - Escrever na saída padrão ou ler entrada de dados padrão.
  - Interação com o usuário, via GUIs, CLIs, etc.
  - Ler ou escrever em arquivos.
  - Ler ou escrever em bancos de dados.
  - Enviar ou receber dados pela rede.
  - etc.

- **Disparo de exceções.** Uma função gera efeito colateral sempre que dispara uma exceção, pois isso pode alterar o fluxo de execução do programa e afetar outras partes do código. O aspecto mais crônico desse efeito colateral é que a exceção pode impedir a resolução do valor de resultado da expressão, violando sua transparência referencial. Isso leva a resultados não-determinísticos, que podem dificultar o entendimento do código e a depuração de erros.

- **Alteração do ambiente de execução.** Uma função gera efeito colateral sempre que altera o ambiente de execução do programa, como por exemplo:
  - Modificar o estado de uma variável de ambiente.
  - Alterar a configuração do sistema ou do ambiente de execução.
  - Modificar o estado de uma biblioteca ou framework utilizado pelo programa.

- **Chamada de módulos com efeitos colaterais.** Uma função pode chamar outra função, método ou módulo que tenha efeitos colaterais. Nesse caso, a função original também gera efeito colateral, mesmo que não o faça diretamente. Por exemplo, em Java:

  - `Math.random()` é uma função que inerentemente gera efeito colateral, pois ela é não-determinística, ou seja, não retorna o mesmo valor para os mesmos argumentos.
  - `System.currentTimeMillis()` é uma função que gera efeito colateral, pois ela retorna o tempo atual do sistema, que é um valor não-determinístico e pode variar a cada chamada.
  - etc.

Problemas gerados por efeitos colaterais Os efeitos colaterais são características indesejáveis em programas declarativos, pois eles tornam o código mais difícil de entender, testar e manter. Entre as desvantagens dos efeitos colaterais, podemos destacar:

- **Imprevisibilidade**. Código com efeitos colaterais são mais difíceis de passar em provas de corretude, pois seu comportamento depende da ordem de execução das instruções e do estado do ambiente de execução em um dado momento. Isso leva a comportamentos não-determinísticos.

- **Dificuldade de teste**. Efeitos colaterais tornam o código mais difícil de testar, pois é necessário considerar o estado do ambiente externo e os efeitos colaterais gerados pela função. Isso pode levar a testes mais complexos e menos confiáveis e, em muitas situações, contextos impossíveis de reproduzir.

- **Dificuldade de depuração**. Efeitos colaterais tornam o código mais difícil de depurar, pois é necessário considerar o estado do ambiente externo e os efeitos colaterais gerados pela função. Isso pode levar a erros difíceis de reproduzir, especialmente em programas concorrentes.

- **Problemas de concorrência**. Em ambientes de execução concorrente, efeitos colaterais podem levar a condições de corrida e outros problemas de concorrência, pois diferentes partes do código podem alterar o estado do ambiente externo ao mesmo tempo. Isso pode levar a resultados imprevisíveis e difíceis de depurar.

- **Redução da componibilidade**. A componibilidade é a capacidade de combinar módulos independentes para criar programas mais complexos. Efeitos

colaterais reduzem a capacidade de contruir software por meio da composição de módulos independentes, pois a resolução de dependências mútuas entre os módulos pode ser afetada pela ordem de execução dos módulos, bem como do ambiente externo, entre outros fatores.

- entre outros.

1.6 Transparência referencial

Definição de transparência referencial A transparência referencial é um princípio fundamental da programação funcional que afirma que uma *expressão* pode ser substituída por seu valor sem alterar o comportamento do programa. Expressões que não satisfazem essa propriedade são consideradas *referencialmente opacas*.

Comandos e transparência referencial Note que o conceito de transparência referencial aplica-se apenas a expressões, e não a comandos. Por isso dizemos que comandos são inerentemente opacos referencialmente, pois eles realizam ações que não podem ser substituídas por seus valores sem alterar o comportamento do programa. Por exemplo, o comando `System.out.println("Hello, World!")` é um comando que imprime uma mensagem na saída padrão, mas não retorna um valor. Se substituirmos esse comando por seu valor (que é `void`), o comportamento do programa será alterado, pois a mensagem não será mais impressa. Funções que retonam `void` (em Java) são considerados como comandos, logo são não transparentes referencialmente. O mesmo pode ser generalizado para outras linguagens, como Python, C, C++, etc.

Exemplos de transparência referencial Vamos considerar alguns exemplos simples de expressões, evitando confusão com funções puras.

- Exemplo de transparência referencial com expressões simples:
  - `int a = 2 + 3;`, pois sempre resulta em 5.
  - `int b = a * 4;`, pois sempre resulta em 20, dado que `a` é 5.
  - `int c = (1 + 2) * (3 + 4);`, pois sempre resulta em 21.
  - `int z = Integer.parseInt("42");`, pois sempre resulta em 42, dado que a string é sempre a mesma.

- Exemplo de opacidade referencial:
  - `int x = (int) (Math.random() * 10);`, pode resultar em valores diferentes a cada execução, logo não é determinística.
  - `long y = System.currentTimeMillis();`, depende do tempo atual, a cada momento gera um resultado diferente.
- Exemplo de efeito colateral (também é opaco referencialmente):
  - `System.out.println("Olá, mundo!");` // Não é transparente referencialmente, pois gera efeito colateral
  - Em C, a expressão `1 + printf("%d", 10) + 2` pode gerar certa confusão, pois `printf` é uma função que gera efeito colateral (imprime na saída padrão), mas retorna a quantidade de caracteres impressos. Mesmo retornando sempre o mesmo valor, essa expressão não é transparente referencialmente, pois gera um efeito colateral no ambiente externo.
  - `Scanner sc = new Scanner(System.in); String s = sc.nextLine();` lê uma string da entrada padrão, o que gera efeito colateral, pois interage com o ambiente externo. Além disso, é não-determinística, pois o valor depende da entrada do usuário.

Resumo A transparência referencial é de suma importância na programação declarativa. Dentro da programação funcional, para que tenhamos transparência referencial em qualquer situação, é requerido que todas as expressões sejam referencialmente transparentes. Isso garante que o comportamento do programa seja previsível, componível e reutilizável.

1.6.1 Funções puras

Definição de função pura Uma função é pura se apresenta as seguintes características:

- **Determinismo**. A função sempre retorna o mesmo resultado para os mesmos argumentos.

- **Ausência de efeitos colaterais**. A expressão não altera o estado do programa ou interage com o ambiente externo.

Toda função pura é referencialmente transparente, pois ao satisfazer as condições de determinismo e ausência de efeitos colaterais, a função sempre pode ser substituída por seu resultado sem prejuízo à corretude do programa. A transparência referencial é uma propriedade desejável em funções puras, atuando como indicador da pureza da função.

Transparência referencial vs. funções puras Embora o conceito de transparência referencial assemelha-se ao de função pura, a transparência referencial é um conceito mais amplo, pois se aplica a qualquer expressão, não apenas a funções. Uma expressão pode ou não conter chamadas de função. Por exemplo, uma expressão matemática como $2 + 2$ é referencialmente transparente, pois sempre retorna o mesmo resultado (4) e não tem efeitos colaterais. A expressão `10 + Math.cos(Math.PI)` também é referencialmente transparente, pois sempre retorna o mesmo resultado (9) e não tem efeitos colaterais. Por outro lado, uma expressão como `System.out.println("Hello, World!")` não é referencialmente transparente, pois gera um efeito colateral (imprime uma mensagem no console) e não retorna um valor.

Se usarmos uma função impura em uma expressão, essa expressão não será referencialmente transparente, pois o resultado da função pode variar dependendo do estado do programa ou do ambiente externo. Por exemplo, a expressão `Math.random()` não é referencialmente transparente, pois retorna um valor diferente a cada chamada e não tem efeitos colaterais. Consequentemente, a expressão `Math.random() + 1` também não é referencialmente transparente, pois o resultado da expressão pode variar a cada chamada.

Exemplos de funções puras Considere o programa 1.15, que define diferentes métodos e funções, algumas puras e outras impuras.

Listagem 1.15: Exemplo de funções puras e impuras

```
1 public class FunctionExamples {
2    private int state = 0;
3
4    // Pure function: returns the sum of two numbers
5    public int add(int a, int b) {
6       return a + b;
7    }
```

```
8
9     // Pure function: returns the factorial of a number
10    public int factorial(int n) {
11       if (n <= 1) return 1;
12       return n * factorial(n - 1);
13    }
14
15    // Impure function: modifies internal state
16    public void incrementState() {
17       this.state++;
18    }
19
20    // Impure function: prints to the console (side effect)
21    public void printMessage(String message) {
22       System.out.println(message);
23    }
24
25    // Pure function: returns the square of a number
26    public int square(int x) {
27       return x * x;
28    }
29
30    // Impure function: access external state and returns a value
31    public int multiply(int x) {
32       return this.state * x;
33    }
34
35    // Impure function: reads the current time (depends on
          external state)
36    public long getCurrentTimeMillis() {
37       return System.currentTimeMillis();
38    }
39
40    // Getter for state (impure if used for testing state changes)
41    public int getState() {
42       return state;
43    }
44 }
```

Vamos analisar cada função do programa:

- `add(int a, int b)`: é uma função pura, pois sempre retorna o mesmo resultado para os mesmos argumentos e não tem efeitos colaterais.
- `factorial(int n)`: é uma função pura, pois sempre retorna o mesmo resultado para os mesmos argumentos e não tem efeitos colaterais. Ela é recursiva, mas isso não afeta sua pureza.
- `incrementState()`: é uma função impura, pois altera o estado interno da classe ao incrementar a variável `state`, que é definida fora do escopo da função. Isso gera um efeito colateral.
- `printMessage(String message)`: é uma função impura, pois realiza uma operação de entrada/saída (I/O) ao imprimir uma mensagem no console. Isso gera um efeito colateral.
- `square(int x)`: é uma função pura, pois sempre retorna o mesmo resultado para os mesmos argumentos e não tem efeitos colaterais.
- `multiply(int x)`: é uma função impura, pois acessa o estado interno da classe (a variável `state`) e retorna um valor que depende desse estado. Isso gera um efeito colateral.
- `getCurrentTimeMillis()`: é uma função impura, pois depende do estado externo (o tempo atual do sistema) e retorna um valor que pode variar a cada chamada. Isso gera um efeito colateral.
- `getState()`: é uma função impura se usada para testar mudanças de estado, pois retorna o valor da variável interna `state`, que pode ter sido alterada por outras funções impuras. No entanto, se usada apenas para obter o valor atual do estado sem modificá-lo, pode ser considerado um efeito colateral de menor gravidade.

Vantagens das funções puras A programação funcional prescreve que todas as funções devem ser puras, ou seja, devem aderir aos princípios de determinismo e ausência de efeitos colaterais. Isso traz diversas vantagens para o desenvolvimento de software, pois elimina os problemas gerados por efeitos colaterais, quais sejam: aumenta a previsibilidade do código, facilita o teste e a depuração, melhora a componibilidade e a reutilização de código, entre outros.

1.7 Recursividade

Na construção de algoritmos, de modo geral, um processo fundamental é a capacidade de efetuar repetições. Tais processos são fundamentais para resolver problemas via iteração, processar quantidades variáveis de dados ou resolver problemas que podem ser divididos em subproblemas menores. Na programação imperativa, repetições são comumente implementadas por meio de estruturas de controle, como laços de repetição (*loops*), que permitem executar um bloco de código várias vezes até que uma condição seja satisfeita. No entanto, do ponto de vista da imutabilidade, os loops são problemáticos, pois eles envolvem mutação de variáveis de controle ou de resultado.

Repetição imperativa Considere o programa 1.16, que calcula o somatório de todos inteiros no intervalo $[a, b]$.

Listagem 1.16: Exemplo de laço de repetição

```
1 public static int sumRangeImperative(int a, int b) {
2    int sum = 0;
3    for (int i = a; i <= b; i++) {
4       sum += i;
5    }
6    return sum;
7 }
```

Como podemos observar, o programa utiliza um laço de repetição `for` para iterar sobre os números no intervalo $[a, b]$ e acumular a soma em uma variável mutável `sum`. Para tal, apoia-se extensivamente em variáveis mutáveis, como `sum` e `i`, que são alteradas a cada iteração do laço. Isso torna o código imperativo, pois o estado do programa é alterado a cada iteração.

Repetição declarativa Em contraste, a programação funcional prescreve o uso de *recursão* como técnica para efetuar repetições. A recursão é um processo em que uma função chama a si mesma para resolver um problema, dividindo-o em subproblemas menores. A recursão é uma técnica poderosa, pois permite expressar algoritmos de forma declarativa, sem a necessidade de mutação de variáveis. Considere o programa 1.17, que resolve o mesmo problema do programa anterior, mas usando recursão.

Listagem 1.17: Exemplo de recursão

```
1 public static int sumRangeNonTailRecursive(int a, int b) {
2    if (a > b) return 0;
3    return a + sumRangeNonTailRecursive(a + 1, b);
4 }
```

Como podemos observar, um algoritmo recursivo é capaz de resolver o mesmo problema sem o emprego de mutações de variáveis. Utiliza apenas expressões e passagem de argumentos para resolver o problema. A função `sumRangeNonTailRecursive` recebe dois argumentos, `a` e `b`, e retorna a soma de todos os inteiros no intervalo $[a, b]$. A função verifica se `a` é maior que `b` e, se for, retorna 0 (caso base). Caso contrário, a função chama a si mesma com o argumento `a + 1` e acumula o resultado com o valor de `a`.

Recursividade e programação funcional Na programação funcional pura, não existe a possibilidade de se utilizar variáveis mutáveis, o que torna o uso das estruturas de controle da programação estruturada inviáveis. Com isso, a recursão se torna a principal técnica para implementar repetições e iterações.

Desvantagem da recursividade Um ponto fraco fundamental dos algoritmos recursivos é que eles podem levar a problemas de desempenho. Quando fazemos chamadas recursivas, o ambiente de execução da linguagem depende de manter chamadas de função em uma pilha. No caso em que processamos grandes volumes de dados, a pilha de recursão pode tornar-se muito profunda, levando a um estouro de pilha (*stack overflow*). Em suma, a complexidade de espaço do algoritmo cresce linearmente com o tamanho da entrada, o que pode levar a problemas de desempenho em casos extremos. Esse é um grande problema de desempenho que precisa ser observado e tratado ao construir algoritmos recursivos, de modo geral, e em linguagens funcionais puras, em específico.

Recursão na cauda A solução dada pelo programa 1.17 é uma forma de recursão não otimizada, portanto considerada ineficiente, pois a pilha de execução terá complexidade de espaço $O(b - a)$ no pior caso, o que inviabiliza o somatório de intervalos muito grandes.

Para lidar com esse problema de escalabilidade, os ambientes de execução de linguagens funcionais modernas, como Haskell, Scala, Clojure, etc., implementam otimizações de recursão, como a otimização de chamada na cauda (*tail call optimization*). A recursão de cauda é uma técnica que permite que o compilador ou interpretador otimize chamadas recursivas para que o tamanho da pilha man-

tenha-se constante. Durante o curso, veremos diferentes técnicas para normalizar nossos algoritmos para que eles apresentem recursão na cauda. Por ora, vamos observar o programa 1.18, que implementa a mesma função de somatório, mas usando recursão na cauda.

Listagem 1.18: Exemplo de recursão na cauda

```
1 public static int sumRangeTailRecursive(int a, int b, int
     accumulator) {
2    if (a > b) return accumulator;
3    return sumRangeTailRecursive(a + 1, b, accumulator + a);
4 }
5 public static int sumRange(int a, int b) {
6    return sumRangeTailRecursive(a, b, 0);
7 }
```

A função `sumRangeTailRecursive` recebe um argumento adicional `accumulator`, que acumula o resultado da soma. A função verifica se `a` é maior que `b` e, se for, retorna o valor acumulado. Caso contrário, a função chama a si mesma com o argumento `a + 1` e acumula o resultado com o valor de `a`. A função `sumRange` é uma função de conveniência que chama a função recursiva com o valor inicial do acumulador igual a 0.

Em linhas gerais, a recursão na cauda estabelece que a chamada recursiva é a última operação a ser executada antes da função retornar um valor. Embora a diferença seja sutil, essa nova versão do algoritmo é em tese mais eficiente, pois permite que a pilha de execução não cresça com o tamanho da entrada. Mais precisamente, se o ambiente de execução da linguagem suportar otimização de recursão na cauda, a complexidade de espaço do algoritmo será $O(1)$, ou seja, constante[4].

1.8 Resumo

A programação funcional é um paradigma declarativo que permite definir programas com base na aplicação e composição de funções puras, evitando estados mutáveis e efeitos colaterais. Entre suas características principais estão: funções de

[4]Em Java, isso não ocorre pois essa linguagem não possui esse tipo de otimização. Esse fato, por si só, é uma boa razão para escolhermos linguagens de programação adequadas quando queremos trabalhar com programação funcional pura.

primeira classe e de ordem superior, imutabilidade, funções puras, recursividade e composição de funções. Os paradigmas de programação evoluem ao longo do tempo, sendo a programação funcional um subconjunto do paradigma declarativo, com suas raízes teóricas no cálculo lambda.

As linguagens funcionais são classificadas em puras (como Haskell) e híbridas, subdividindo estas últimas em foco funcional (e.g., Scala. Scheme, Clojure) e foco imperativo/OO (e.g., JavaScript, Python). O paradigma de programação funcional vem sendo desenvolvido desde os anos 1930, apresentando uma crescente adoção nas décadas recentes, impulsionada por áreas como Big Data, computação em nuvem e a incorporação de seus conceitos em linguagens majoritárias multiparadigma.

Aviso de licença de uso