PARADIGMA E DESIGN DE CÓDIGO COM PROGRAMAÇÃO FUNCIONAL EM JAVA

Guilherme de Cleva FARTO
guilherme.farto@gmail.com

Paulo Cesar ROMANO
paulo-romano_133@hotmail.com

RESUMO: Com o avanço da tecnologia e a contínua necessidade de desenvolver *softwares* cada vez mais robustos, torna-se essencial o cuidado e o planejamento durante o seu desenvolvimento, visando a qualidade, legibilidade e facilidade na compreensão código por meio dos recursos que uma linguagem de programação pode oferecer. O objetivo deste trabalho de pesquisa é explorar os conceitos da programação funcional em Java e explorar, por meio de implementações, as características e diferentes sintaxes deste paradigma em relação ao paradigma imperativo.

PALAVRAS-CHAVE: Java, Código Limpo; Programação Funcional, API de Stream;

ABSTRACT: With the advancement of technology and the continuous need to develop increasingly robust software, care and planning during its development becomes essential, aiming at the quality, readability and ease of understanding code through the resources that a programming language can offer. The objective of this research work is to explore the concepts of functional programming in Java and explore, through implementations, the characteristics and different syntaxes of this paradigm in relation to the imperative paradigm.

KEYWORDS: Java; Clean Code; Functional Programming; Stream’s API;

1. Introdução

Paradigmas na programação definem a forma padronizada em que um problema será solucionado por meio de uma linguagem ou programa. Por meio deles é delimitado a sintaxe e funcionalidades de uma linguagem, que consequentemente delinearão a estrutura e organização do código (TUCKER; NOONAN, 2010, p 3).

A partir da versão 8 do Java, lançada em 2014, foi introduzido o paradigma de programação funcional à linguagem, tendo em vista a modelagem de "um problema computacional como uma coleção de funções matemáticas, cada uma com um espaço de entrada (domínio) e resultado (faixa)" (TUCKER; NOONAN, 2010, p 4). Este paradigma preserva a imutabilidade dos dados, abstrai certas responsabilidades e reduz linhas de código que na programação imperativa eram necessárias, bem como o acoplamento de funções. Tudo isso beneficia tanto a consistência do código quanto dos dados (PEREIRA, 2015).

2. Paradigma Imperativo

A fundamentação do paradigma imperativo está relacionada ao conceito de abstração matemática da máquina de Turing "que corresponde ao conjunto de funções computáveis" (MELO; SILVA, 2014, p 158).

Tendo o seu surgimento na década de 1940 graças a John von Neumann e seus colaboradores, este paradigma trouxe consigo descobertas e uma maior versatilidade para a computação, permitindo que com isso programas fossem capazes de armazenar suas instruções e os valores dos dados na memória (SILVA; LEITE; OLIVEIRA, 2019, p 34).

No âmago da arquitetura do modelo de von Neumann-Eckert está a base do paradigma imperativo que traz a ideia de atribuição, ou seja, "alterar o valor de um local de memória e destruir seu valor anterior" (TUCKER; NOONAN, 2010, p 278).

> Já que elas surgiram do modelo de von Neumann-Eckert, todas as linguagens imperativas incluem a atribuição como um elemento central. Além disso, elas suportam declarações de variáveis, expressões, comandos condicionais, laços e abstração procedural. Declarações atribuem nomes a locais de memória e associam tipos aos valores armazenados (TUCKER; NOONAN, 2010, p 278).

3. Paradigma Estrutural

Este modelo de programação surgiu a partir da motivação por meio de críticas ao comando *go to*, visto que o seu uso amplamente utilizado pelos programadores "gerava um número considerável de programas praticamente impossíveis de serem entendidos, e

a consequência disso era a dificuldade de manutenção de programas" (SILVA; LEITE; OLIVEIRA, 2019, p 45).

A fim de corrigir os problemas causados pelo comando *go to*, este novo paradigma consistia na utilização de três estruturas de controle, sendo elas a "sequência" que "implementa os passos necessários para a descrição de qualquer problema"; a "seleção" que diz respeito a execução de um fluxo de processamento a partir da validação de uma condição (*if* e *switch*); a "iteração" que permite a repetição de códigos contidos em um escopo enquanto a condição for satisfeita, como é o caso do *while* e *for* (SILVA; LEITE; OLIVEIRA, 2019, p 46).

4. Paradigma Funcional

"A característica essencial da programação funcional é que a computação é vista como uma função matemática mapeando entradas e saídas". Neste modelo não há noção de estado pois as instruções estão implícitas na API (TUCKER; NOONAN, 2010, p 362).

Este modelo de programação traz consigo uma nova forma de pensar o fluxo do algoritmo com a introdução do paradigma declarativo e assegura a ausência de efeitos colaterais dentro de um fluxo com a imutabilidade da variável, eliminando assim os *side-effects* e obtendo, por fim, códigos mais concisos (PEREIRA, 2015).

A imutabilidade consiste na inalteração do valor até o final do processo, uma vez que este valor é atribuído a uma função. Porém, devido a isso, há uma tendência no maior consumo de memória, pois a imutabilidade consiste em trabalhar com cópias e não referências (SILVA; LEITE; OLIVEIRA, 2019, p 136).

Outro conceito presente neste paradigma é a transparência referencial que se define quando o valor de uma função "depende apenas dos valores de seus argumentos, ou seja, a função sempre produz o mesmo resultado a partir dos mesmos parâmetros" (SILVA; LEITE; OLIVEIRA, 2019, p 136).

O paradigma funcional possui duas vertentes, sendo elas denominadas de puras e impuras. A primeira se refere as linguagens funcionais que "eliminam a noção de célula de memória de uma variável em favor da noção matemática; isto é, uma variável dá nome a expressão imutável, que também elimina o operador de atribuição". Já a segunda, retêm alguma forma de operador de atribuição, e, portanto, são impuras (TUCKER; NOONAN, 2010, p 363).

Em linguagens funcionais, há uma distinção na forma como uma função é avaliada, podendo esta ser rápida – conhecida também como chamada por valor –, que "refere-se à estratégia de avaliar todos os argumentos para uma função no instante de sua chamada",

o que acaba por contribuir em sua eficiência, enquanto na avaliação lenta o argumento para 9 a função não é avaliado enquanto ele não for necessário, no entanto, esta permite a implementação de certas funções que não são possíveis em linguagens rápidas (TUCKER; NOONAN, 2010, p 365).

Este paradigma gera um código completo e autossuficiente conforme o comportamento passado, interpretando-o como expressões matemáticas. As vantagens em seu uso envolvem a visualização dos programas como funções uniformes, notação concisa, facilidade nos testes e na busca por bugs, uso de gerenciamento de memória automático, tratamento das funções como dados, grande flexibilidade e semântica simples (KEOMA, 2018).

5. Proposta de Desenvolvimento do Trabalho

5.1. Objetivos

Esta pesquisa de iniciação científica tem como objetivo principal explorar as funcionalidades da API de *Stream* do Java e implementar estes recursos disponíveis na API paralelamente ao paradigma imperativo, a fim de explicitar as possíveis diferenças entre estas duas abordagens.

5.2. Tecnologias e Recursos Utilizados

5.2.1. Java

A linguagem Java, herdando o legado das linguagens C e C++, foi criada por James Gosling, Patrick Naughton, Chris Warth, Ed Frank e Mike Sheridan na Sun Microsystems, tendo inicialmente o nome “Oak”, até que em 1995 foi renomeada como “Java” (SCHILDT, 2015, p 3).

O *bytecode*, gerado pelo compilador Java, é um conjunto de instruções com grande otimização, sendo projetado para ser executado pela JVM (Java *Virtual Machine*). Isso facilitou a execução de programas Java em diversos sistemas, visto que dispensou a necessidade de recompilar o programa, sendo necessário somente ter instalado a JVM da respectiva plataforma para que o sistema possa ser executado por meio do mesmo *bytecode* (SCHILDT, 2015, p 6).

5.2.2. API de Stream do Java 8

Com o advento da versão 8 do Java, feito pela Oracle em 2014, foram introduzidas mais de 80 novas funcionalidades com alterações significativas na máquina virtual (JVM), causando impactos na forma como o desenvolvedor escreve o código (SILVA, 2016).

Uma das *features* introduzidas são as Expressões Lambda, que trouxeram mudanças na sintaxe com a incorporação do paradigma funcional, oferecendo maior versatilidade e facilidade em tarefas que antes demandavam maior complexidade e muitas linhas de código, tornando possível também a passagem de comportamentos, ou seja, funções como argumentos de métodos (SILVA, 2016).

Uma *Stream* pode ser definida "como uma sequência de elementos de uma fonte de dados que suporta operações de agregação". Sua abordagem de trabalho envolve "a conversão de uma coleção, *array* ou recursos I/O (entrada e saída de dados) em uma *Stream*, o processamento dos dados e a devolução dos dados em uma nova coleção ou valor reduzido (*map-reduce*)" (AMORIM, 2015).

Desenvolvida sob o pacote java.util.stream, suas operações podem ser classificadas como intermediárias quando retornam uma nova *Stream*, fazendo com que seja possível o encadeamento múltiplo destas operações; e as terminais, que tem por finalidade fechar o processo das operações que a precederam (SILVA, 2017).

Ademais, operações relacionadas a *Streams* possuem duas características que as diferem das operações sobre coleções. São elas a Pipeline, composta de uma ou mais operações intermediárias que diz respeito àquelas que retornam novas *Streams*, possibilitando "criar uma cadeia de operações que formam um fluxo de processamento", encerrando-se com uma operação terminal; e a Iteração Interna, que encapsula na API a responsabilidade do modo como ocorre a iteração através de métodos como o *map, forEach, filter*, entre outros (SILVA, 2017).

Dentro do escopo das operações presentes na API de *Stream* há particularidades importantes de serem destacadas, sendo elas:

5.2.2.1. Lazy Evaluation

Estas operações trabalham de forma preguiçosa, ou seja, não executam nenhum processamento até que uma operação terminal seja invocada (URMA; FUSCO; MYCROFT, 2019, p 94).

Através dessa característica, é possível otimizar as operações de forma que não haja processamento desnecessário, a fim de obter somente o que foi solicitado pela operação

terminal, como é o caso das operações que retornam um *Optional* (SILVEIRA; TURINI, p 56-57).

Operações terminais como o *findAny* são escritos de forma inteligente, "analisando as operações invocadas anteriormente e percebendo que não precisa filtrar todos os elementos da lista para pegar apenas um deles que cumpra o predicado". No entanto, esta técnica não garante que será possível aplica-la em qualquer contexto, pois depende das operações contidas no pipeline (SILVEIRA; TURINI, p 58).

5.2.2.2. Stateless

Stateless são operações que não armazenam o estado do elemento anterior para manipulação dos elementos subsequentes que serão processados, ou seja, cada elemento independe do que acontece com os demais (SILVA, 2016).

Sendo assim, operações como *map* e *filter* são exemplos de *stateless*, porém a inexistência de um estado pressupõe que, ou não há tal presença na expressão atribuída para o método ou o método não possui um estado mutável interno (URMA; FUSCO; MYCROFT, 2019, p 115).

5.2.2.3. Stateful

Estas operações "podem incorporar o estado do elemento processado anteriormente no processamento de novos elementos" (SILVA, 2016), como é o caso de métodos como o *reduce*, *sum* e *max* que possuem um estado que vai acumulando o resultado obtido por cada elemento a fim de gerar o resultado final (URMA; FUSCO; MYCROFT, 2019, p 116).

Ademais, operações *stateful* "podem precisar processar todo o *stream* mesmo que sua operação terminal não demande isso", o que acarretaria em perda de performance com processamento desnecessário (SILVEIRA; TURINI, p 59).

5.2.2.4. Short-circuiting

Por meio desta otimização tornou possível a API obter o resultado final sem que sejam processados todos os elementos (SILVA, 2016). Com isso, métodos como *allMatch* e *findFirst* interrompem o fluxo de processamento assim que a sua condição é ou não atendida (URMA; FUSCO; MYCROFT, 2019, p 109).

5.2.2.5. Reduction

Operações de redução trabalham com a finalidade de reduzir um conjunto de valores de um fluxo de dados de uma *stream* para um único valor, sendo classificados, portanto, como operações terminais (URMA; FUSCO; MYCROFT, 2019, p 111).

Fazendo parte deste tipo de operação encontram-se métodos: *average*, *count*, *min*, *max* e o sum, tendo alguns deles o retorno de um *Optional* para casos em que não há valor a ser retornado (SILVEIRA; TURINI, p 59).

5.2.2.6. Método collect

Classificado como operação terminal, o método *collect* "possibilita coletar os elementos de uma *stream* na forma de coleções, convertendo uma *stream* para os tipos *List*, Set ou Map" (SILVA, 2016), além de possuir métodos como o mapping, *collectingAndThen* e *grouppingBy* que concedem a combinação de elementos e o *joining* que tem a função de concatenar elementos para uma String (SUBRAMANIAM, 2014, p 54).

5.2.2.7. Parelelismo

A API também dá suporte à paralelização através da chamada do método *parallelStream* que já contém toda a lógica de baixo nível para a sua execução, sendo a cargo do desenvolvedor somente construir o fluxo do pipeline (SILVA, 2016).

Consequentemente fica a cargo da API "decidir quantas threads deve utilizar, como deve quebrar o processamento dos dados e qual será a forma de unir o resultado final em um só" (SILVEIRA; TURINI, P 78).

No entanto, há de se ter mais cautela, pois não havendo uma grande quantidade de elementos pode ocasionar em um overhead, que é "decorrente do processamento de tarefas adicionais geradas pela paralelização" (SILVA, 2016).

5.3. Funções Lambdas em Java

Em Java a sintaxe de uma função lambda é definida pelo argumento e corpo da função. Além disso, estas funções, podendo ou não possuir parâmetros, são capazes de omitirem a utilização das chaves "{}" para delimitação do seu escopo, caso no corpo da função haja somente um comando. Do mesmo modo, nesse contexto é possível suprimir o comando *return* caso a função precise retornar algo (SANTANA, 2015).

Como mais uma característica na sintaxe da construção de uma função lambda, o Java 8 trouxe consigo a Inferência de Tipo, desobrigando o programador a definir qual o tipo do parâmetro informado, deixando a cargo do compilador inferir, em tempo de compilação,

qual o tipo que se espera. Consequentemente esta abordagem deixará o código mais limpo (SUBRAMANIAM, 2018).

5.4. Method References

Method References é uma sintaxe alternativa da representação de uma função lambda que trabalha com a chamada de um método já existente por meio do seu nome ao invés de montar a expressão correspondente (URMA; FUSCO; MYCROFT, 2019, p 65).

A sua usabilidade engloba referência a métodos de instância, estáticos e parametrizados, porém isso só é realizável para métodos que esperam uma Interface Funcional como parâmetro (SUBRAMANIAM, 2014, p 25).

5.5. Interfaces Funcionais

Interfaces Funcionais em Java são interfaces que possibilitam a escrita de códigos utilizando a sintaxe de expressões lambda. Esta abordagem garantiu que fosse mantida a integridade com as versões anteriores da linguagem (SUBRAMANIAM, 2017).

Para uma interface ser considerada funcional ela deve conter um e somente um método abstrato, podendo possuir, no entanto, métodos default e estáticos. Contudo, há uma ressalva para o caso deste método abstrato estar na classe *Object* como método público (SUBRAMANIAM, 2017).

Ademais, por convenção da linguagem e para que o compilador possa reconhecer que uma determinada interface é de fato uma interface funcional, ela disponibiliza a anotação *@FunctionalInterface*, de modo que seja gerado algum erro de compilação caso os requisitos para a criação desta interface não sejam providos (CARVALHO, 2018).

Ao achar uma expressão lambda no código, o compilador, através da extração do código lambda para uma função tipada, irá encontrar a interface funcional correspondente a função extraída e implementar o método abstrato que esta interface possui. A chamada deste método se dá de forma dinâmica, por meio do *invokedynamic*, disponível a partir da versão 7 da linguagem (REIS, 2017).

5.6. Trabalhos Futuros

Para trabalhos futuros poderão ser explorados mais recursos da API de *Stream*, demonstrando também a abordagem paralela, as diferenças entre esta e a *stream* serial e as possíveis vantagens e desvantagens que o seu uso pode causar em uma aplicação.

6. Anexos

Projeto contendo implementações utilizando os paradigmas imperativo e funcional. Disponível em: <https://github.com/paulocromano/paradigmas-imperativo-funcional-java>.

7. Resultados

Com base nas implementações presentes no projeto, com o intuito de evidenciar como se dá o processamento dos dados por meio dos paradigmas imperativo e funcional, e como este permite diferentes sintaxes durante a construção do código, pode-se notar a dessemelhança entre estes dois, sobretudo na complexidade, legibilidade e tamanho do código. Quanto mais operações os dados eram submetidos, maior era a diferença que estes dois modelos de programação apresentavam.

Portanto, o paradigma funcional se mostrou mais prático e flexível por possuir diversas funcionalidades e possibilitar o encadeamento de operações. Tudo isso também se deve a abstração de toda a parte lógica, ou seja, "como" a linguagem iria processar os comportamentos que eram passados via lambda ou *Method Refrences*.

8. Conclusões

Com o desenvolvimento de *softwares* mais robustos e complexos torna-se imprescindível a escrita de códigos limpos, legíveis e bem estruturados, seguindo padrões e boas práticas, a fim de facilitar a manutenção e implementação de novas funcionalidades.

Dito isto, esta pesquisa propôs explorar alguns dos recursos disponíveis na API de *Stream*, as diferentes sintaxes que a linguagem Java possibilita utilizar para a criação de expressões lambdas por meio das Interfaces Funcionais e como este modelo de programação se diferencia do paradigma imperativo, visto que a preocupação desse se dá em "o quê" o programador deve fazer ao invés de "como fazer".

9. Referências Bibliográficas

CARVALHO, M. **Java 8 – Functional Interfaces – Tornando o Java mais legal**. 2018. Disponível em: <https://medium.com/@mvalho/java-8-functional-interfaces-tornando-ojava-mais-legal-72401462d0e2>. Acesso em 16 set. 2021.

KEOMA D. **As vantagens da Programação Funcional (PF)**. 2018. Disponível em: <https://administradores.com.br/artigos/as-vantagens-da-programacao-funcional-pf>. Acesso em 10 out. 2021.

MELO, A.; SILVA, F. **Princípios de linguagens de programação**. 3ª ed. São Paulo: Editora Edgard Blücher, 2014.

PEREIRA, R. **Programação funcional com Java**. 2015. Disponível em: <https://www.devmedia.com.br/programacao-funcional-com-java/32176>. Acesso em 25 set. 2021.

REIS, J**. Por trás da programação funcional do JAVA 8**. 2017. Disponível em: <https://medium.com/dev-cave/por-trás-da-programação-funcional-do-java-8-bd8c0f172d45>. Acesso em 18 set. 2021.

SCHILDT, H. **Java para iniciantes**. 6ª ed. Tradução de Aldir José Coelho Corrêa da Silva. Porto Alegre: Bookman, 2015.

SILVA, C. **Java 8: Iniciando o desenvolvimento com a Streams API**. 2016. Disponível em: <https://www.oracle.com/br/technical-resources/articles/java-stream-api.html>. Acesso em 29 set. 2021.

SILVA, C. **Java 8: Iniciando o desenvolvimento com a Streams API**. 2017. Disponível em: <https://www.infoq.com/br/articles/java8-iniciando-desenvolvimento-com-a-streamsapi/>. Aceso em 24 out. 2021.

SILVA, F.; LEITE, M.; OLIVEIRA, D. **Paradigmas de programação**. Porto Alegre: Sagah, 2019.

SILVEIRA, P.; TURINI, R. Java 8 Prático: **Lambdas, Streams e os novos recursos da linguagem**. Casa do Código.

SUBRAMANIAM, V. **Function Programming in Java: Harnessing the Power of Java 8 Lambda Expressions**. Texas: The Pragmatic Bookshelf. 2014.

SUBRAMANIAM, V**. Interfaces funcionais: Saiba como criar interfaces funcionais customizadas e entenda por que você deve usar integrações sempre que possível**.

2017. Disponível em: <https://developer.ibm.com/br/articles/j-java8idioms7/>. Acesso em: 16 set. 2021.

SUBRAMANIAM, V. **O Java conhece seu tipo**. 2018. Disponível em: <https://developer.ibm.com/br/articles/j-java8idioms8/>. Acesso em 16 set. 2021.

TUCKER, A.; NOONAN R. **Linguagens de Programação: Princípios e Paradigmas**. 2ª ed. Tradução de Mario Moro Fecchio e Acauan Fernandes. São Paulo: AMGH. 2010.

URMA, R.; FUSCO M.; MYCROFT, A. **Modern Java in Action: Lambdas, streams, functional and reactive programming**. New York: Manning, 2019.