UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS I
CENTRO DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO
CURSO BACHARELADO EM COMPUTAÇÃO

BRUNO SANTOS DE MIRANDA

EXPRESSÕES LAMBDA NO JAVA SE 8: UM ESTUDO INTRODUTÓRIO

CAMPINA GRANDE
2019

BRUNO SANTOS DE MIRANDA

EXPRESSÕES LAMBDA NO JAVA SE 8: UM ESTUDO INTRODUTÓRIO

Monografia apresentada ao curso de bacharelado em ciência da computação da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de Bacharel em Computação.

Área de concentração: Linguagem de Programação.

Orientador: Prof. Me. Edson Holanda Cavalcante Júnior

CAMPINA GRADE
2019

M672e Miranda, Bruno Santos de.
Expressões Lambda no Java SE 8 [manuscrito] : um estudo introdutório / Bruno Santos de Miranda. - 2019.
66 p. : il. colorido.
Digitado.
Trabalho de Conclusão de Curso (Graduação em Computação) - Universidade Estadual da Paraíba, Centro de Ciências e Tecnologia , 2020.
"Orientação : Prof. Me. Edson Holanda Cavalcante Júnior , Departamento de Computação - CCT."
1. Paradigma funcional. 2. Linguagem Java. 3. Expressões Lambda. 4. Programação funcional. I. Título
21. ed. CDD 005.13

Elaborada por Giulianne M. Pereira - CRB - 15/714 BC/UEPB

BRUNO SANTOS DE MIRANDA

EXPRESSÕES LAMBDA NO JAVA SE 8: UM ESTUDO INTRODUTÓRIO

Monografia apresentada ao curso de bacharelado em ciência da computação da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de Bacharel em Computação.

Área de concentração: Linguagem de Programação.

Aprovada em 11 de Dezembro de 2019.

Prof. Me. Edson Holanda Cavalcante Júnior
DC-UEPB - Orientador

Prof. Dr. Wellington Candeia de Araújo
DC-UEPB - Examinador

Prof. Dr. Alysson Filgueira Milanez
DC-UEPB - Examinador

AGRADECIMENTO

Agradeço primeiramente à Deus, que me deu forças e saúde para realizar esse trabalho. Aos meus pais e meus irmãos, que sempre me apoiaram em minha trajetória estudantil. A minha esposa, que esteve presente do início ao fim, sempre me motivando e compreendendo. Ao meu orientador, pela oportunidade, compreensão, disponibilidade, sugestões, paciência e toda ajuda dada durante a realização deste trabalho. Aos colegas de curso que estiveram presentes durante todo o curso e as pessoas que me ajudaram direta ou indiretamente a chegar até aqui.

RESUMO

Este trabalho apresenta aspectos da programação funcional (Expressões Lambda), inseridas na linguagem Java SE 8. Essa introdução proporcionou ganho na estrutura do código e facilidade no uso da programação paralela. De forma geral, o trabalho tem o papel de difundir a utilização da programação funcional, utilizando a linguagem Java, e verificar se existem diferenças significativas em relação ao tempo de execução quando comparado a soluções convencionais. A partir de duas implementações para alguns problemas conhecidos da computação, sendo uma utilizando o paradigma orientado a objetos e outra utilizando as expressões lambda (API do Java SE 8), foi realizada uma análise quantitativa de métricas associadas aos problemas. Observou-se que quando utilizado as Expressões Lambda, nos problemas abordados, não apresentou grandes diferenças em relação ao tempo de execução, quando comparada com métodos de programação convencionais utilizando a linguagem Java. Com a adição deste recurso, a linguagem Java torna-se flexível e robusta, proporcionando ao programador outro formato de programação. Contudo, são necessários conhecimentos mínimos da programação funcional e a compreensão do momento adequado de utilizar a API. Como trabalho futuro foi sugerido a investigação da utilização das expressões lambda no processamento em paralelo.

Palavras-Chave: Paradigma funcional; Linguagem Java; Expressões Lambda.

ABSTRACT

This work presents aspects of functional programming (Lambda Expressions), inserted in the Java SE 8 language. This introduction provides gain in code structure and ease of use of parallel programming. In general, the job or role of spreading the use of functional programming, the use of the Java language and verifying if there are significant differences in relation to the execution time when executing the solutions used. From two implementations for some known computing problems, using an object-oriented paradigm and another using lambda expression (Java SE 8 API), a quantitative analysis of the problem-controlled metrics was performed. Observe that the use of Lambda Expressions in the problems addressed does not differ greatly in runtime compared to programming methods using the Java language. With the addition of this feature, the Java language becomes flexible and robust, applying another programming format to the programmer. However, working programming skills and understanding of the proper timing to use an API are required. As future work, it was suggested to investigate the use of lambda expressions in parallel processing.

Keywords: Functional Paradigm; Java language; Lambda Expressions

LISTA DE ILUSTRAÇÕES

Figura 1 - Hierarquia de herança simples. 20
Figura 2 - Representação da sintaxe de uma Expressão Lambda Java SE 8. 27
Figura 3 - VisualVM, tela inicial. 35
Figura 4 -VisualVM, aba overview 35
Figura 5 – VisualVM, aba Monitor. 36
Figura 6 - VisualVM, aba Threads 37
Figura 7 - VisualVM, Sampler CPU. 37
Figura 8 - VisualVm, Sampler memória. 38
Figura 9 - VisualVm, aba Profiler. 38
Figura 10 - Problema 1 (método convencional) 42
Figura 11 - Problema 1 (método lambda) 42
Figura 12 - Problema 2, Fibonacci recursivo. 45
Figura 13 - Problema 2, Fibonacci recursivo. 46
Figura 14 - Problema 3. Fatorial iterativo. 47
Figura 15 - Problema 3. Fatorial iterativo. 49
Figura 16 - VisualVM, tela inicial. 52
Figura 17 - VisualVM, aba Overview. 53
Figura 18 - VisualVM, aba Sampler. 53
Figura 19 - VisualVM, aba Monitor. 54
Figura 20 - VisualVM, Snapshot demonstração. 54
Figura 21 - VisualVM, imagem Thread CPU time. 55
Figura 22 - Comparação dos Snapshot (Problema 1, Teste 3). 56
Figura 23 - Comparação dos Snapshot (Problema 2, Teste 2) 59
Figura 24 – Snapshot, método lambda Teste 3, Problema 3 60

LISTA DE QUADROS

Quadro 1 - Interfaces funcionais básicas do pacote (java.util.function). 28
Quadro 2 - Operações Stream terminais 31
Quadro 3 - Operações Stream intermediárias 31
Quadro 4 - Tipos de Method Reference. 33
Quadro 5 - Configuração Notebook 39

LISTAS DOS EXEMPLOS DOS CÓDIGOS

Listagem 1 - Utilizando Classe Anônima. 25
Listagem 2 - Utilizando Expressões Lambda 26
Listagem 3 - Interface Funcional Validador 29
Listagem 4 - Classe Principal, utilizando a interface Validador com Classe Anônima. 29
Listagem 5 - Classe Principal, utilizando a interface Validador com Expressão Lambda. 30
Listagem 6 - Sem Stream, forma convencional. 32
Listagem 7 - Com Stream, utilizando Lambda 32
Listagem 8 - Medindo o tempo de execução de um algoritmo 41
Listagem 9 - Método: SomaTradicional(). 43
Listagem 10 - Método: SomarElementosLambda(). 43
Listagem 11 - Problema 2 (método convencional) 45
Listagem 12 - Problema 2 (método lambda) 46
Listagem 13 - Problema 3 (método convencional) 48
Listagem 14 - Problema 3 (método lambda) 48

LISTA DAS TABELAS

Tabela 1- Resultados obtidos do método (Problema 1).....56
Tabela 2 - Resultados obtidos da aplicação do Problema 1 57
Tabela 3 - Quantidade de linhas (Problema 1) 57
Tabela 4 – Resultado obtido do método (Problema 2) 58
Tabela 5 - Resultados obtidos, aplicação (Problema 2).....58
Tabela 6 - Quantidade de linhas (Problema 2) 59
Tabela 7 - Resultados obtidos, método (Problema 3).....60
Tabela 8 - Resultados obtidos, aplicação (Problema 3).....60
Tabela 9 - Quantidade de linhas (Problema 3) 61

LISTA DE ABREVIATURA E SIGLAS

API	Application Programming Interface
AWT	Abstract Window Toolkit
CPU	Central Processing Unit
GC	Garbage Collector
IDE	Integrated Development Environment
JDK	Java Development Kit
JRE	Java Runtime Environment
JVM	Java Virtual Machine
PF	Programação Funcional
POO	Programação Orientada a Objetos
WWW	World Wide Web

SUMÁRIO

1 INTRODUÇÃO 13

1.1 METODOLOGIA 15

2 FUNDAMENTAÇÃO TEÓRICA 16

2.1 PROGRAMAÇÃO FUNCIONAL 16

2.2 PROGRAMAÇÃO ORIENTADA A OBJETOS 18

2.2.1 Abstração 18

2.2.2 Encapsulamento 19

2.2.3 Herança 19

2.2.4 Polimorfismo 20

2.2.5 Método e Atributos 21

2.2.6 Classes e Objetos 21

2.2.7 Vantagem e Desvantagem 22

2.3 A LINGUAGEM JAVA 23

2.3.1 Simples, orientado a objeto e familiar 23

2.3.2 Robusto e Seguro 23

2.3.3 Arquitetura Neutra e Portátil 24

2.3.4 Alto Desempenho 24

2.3.5 Interpretada, Dinâmica e Encadeada 24

2.4 JAVA SE 8 24

2.4.1 Expressões Lambda Java SE 8 25

2.4.2 Sintaxe 26

2.4.3 Utilizando expressões lambda 28

2.5 PROFILING 33

3 DESENVOLVIMENTO DAS APLICAÇÕES 39

3.1 FERRAMENTAS UTILIZADAS 39

3.2 PROBLEMAS ABORDADOS 39

3.3 IMPLEMENTAÇÃO DOS PROBLEMAS 40

3.3.1 Problema 1 41

3.3.2 Problema 2 44

3.3.3 Problema 3 47

4 MÉTRICAS AVALIADAS 50

4.1 TEMPO DE EXECUÇÃO 50

4.2 QUANTIDADE DE LINHAS NO CÓDIGO 50

5 TESTES 51

5.1 PASSOS PARA A REALIZAÇÃO DO EXPERIMENTO 52

6 RESULTADOS 56

6.1 PROBLEMA 1 56

6.2 PROBLEMA 2 58

6.3 PROBLEMA 3 59

6.4 ANÁLISE DOS RESULTADOS 61

7 CONCLUSÃO 63

REFERÊNCIAS 64

1 INTRODUÇÃO

Em 2014 a Oracle lançou a versão 8 do Java SE, trazendo novas funcionalidades, dentre elas o suporte à Programação Funcional (PF) com a adição das Expressões Lambda à sua programação. As especificações do Projeto Lambda são encontradas em **JSR 335: Expressões Lambda para a linguagem de programação Java** (ORACLE, 2014)**.** De acordo com Carvalho (2015, p. 19), as expressões lambda foram adicionadas para sanar um problema antigo da linguagem Java.

Para a compreender este fato, vamos retornar para a primeira versão do Java, para criar interface gráfica foi sugerido o *Abstract Window Toolkit (AWT[1])* que, na sua primeira versão era necessário sobrescrever métodos herdados de suas classes. Desta forma, os eventos gerados pelo usuário, como um clique de um botão, eram obrigatoriamente tratados. No entanto, esta forma de construção de interface gráfica não era uma boa prática de programação devido ao forte acoplamento da classe-pai. Isso foi logo corrigido na versão 1.1 pelas interfaces *Listeners.*

Neste sentido, foram introduzidas duas modificações: a interface *ActionListener* que substitui a sobrescrita de método da herança de classe, e o recurso das Classes Internas Anônimas (*Anonymous Inner Class)*. Esta característica da linguagem solucionava o problema, no entanto, ela foi utilizada em outros momentos, como a criação de instâncias do tipo *Runnable,* com o intuito de executar pedaços de códigos em paralelo utilizando *Threads***.**

Com o passar dos anos e a evolução da linguagem, a Oracle percebeu que o método das Classes Internas Anônimas precisava de melhoria, buscando um código mais limpo e simples. A solução encontrada foi a adição de características da PF, que torna mais simples e prática a criação de trechos de códigos. Esta funcionalidade é conhecida como “Expressões Lambda”. Para Tavares e Caldas (2017, p. 18), a utilização de expressões lambda apresenta as seguintes vantagens em relação as Classes Internas Anônimas: redução de código, legibilidade, organização e facilidades ao utilizar a programação paralela (utilizando os *Streams).*

Sendo assim, o ganho com a utilização das expressões lambda será só estrutural? Será que existe ganho em tempo de execução? Existem outros benefícios para a linguagem Java, quando utilizamos características da PF? Buscando responder estas perguntas, propomos a investigação dos ganhos obtidos com a utilização das expressões lambda que estão presentes na API do Java SE 8.

---

[1] Abstract Window Toolkit (AWT) biblioteca para construção de interface gráfica para desktop.

Antes de responder estas questões apresentaremos um estudo sobre os paradigmas mencionados nesta pesquisa e a utilização das expressões lambda. No decorrer do trabalho apresentaremos alguns exemplos, fortalecendo as afirmações dos autores em relação do ganho estrutural quando utilizado este recurso.

Para responder às perguntas mencionadas, desenvolveremos aplicações que compararão duas soluções para problemas conhecidos da computação, os quais apresentaremos posteriormente. As soluções foram implementadas de duas maneiras: da forma convencional e utilizando expressões lambda. Com ajuda de uma ferramenta (*Profiling)*, capturamos informações sobre o tempo de execução do método, tempo de execução total da aplicação e a quantidade de linhas de código.

Utilizamos métricas quantitativas, já que métricas qualitativas (legibilidade, simplicidade e organização do código entre outras) não fizeram parte do nosso estudo, visto que autores, como Tavares e Caldas (2017) e Teixera Junior (2014), demonstraram isso em seus trabalhos. O nosso foco é difundir os conceitos e características da PF utilizando linguagem Java, destacando as vantagens, desvantagens e o momento adequando para utilizá-la. Outro ponto é verificar se existem perdas significativas em relação ao tempo de execução, quando utilizadas expressões lambda em uma implementação. Ao final os resultados serão apresentados e discutidos.

1.1 METODOLOGIA

A metodologia seguida neste trabalho foi desenvolvida em 4 fases: pesquisa e coleta de dados sobre o assunto abordado, leitura e separação dos conteúdos encontrados, desenvolvimento das aplicações e realização dos testes com a apresentação dos resultados obtidos.

Na primeira fase foram realizadas pesquisas em bibliotecas digitais como: ACM (Association for Computing Machinery), UnB (Universidade de Brasília), BDTD (Biblioteca Digital Brasileira de Teses e Dissertações); livros; plataformas *online*; revistas e documentação Java. Neste quesito foram coletados documentos com assuntos correlatos à pesquisa.

Na segunda fase foram analisados os documentos adquiridos, obtendo um melhor entendimento dos assuntos abordados. Ainda nesta etapa, fizemos a separação dos artefatos coletados para utilizar como referência deste trabalho.

A terceira fase (desenvolvimento das aplicações) trata-se do estudo de caso da pesquisa, que foi dividida em duas etapas: seleção e implementação dos problemas. Na seleção, foram levados em consideração modelos computacionais conhecidos e adequados às expressões lambda do Java SE 8. Na Implementação, foram desenvolvidas as aplicações dos casos de teste relacionados aos problemas. Para cada problema foram feitas duas implementações: uma utilizando a programação orientada a objetos, que chamaremos de “método convencional”; e outra utilizando as expressões lambda, que chamaremos de “método lambda”. Os detalhes da implementação são abordados na Seção 3.

Na última fase, foram realizados os testes nas implementações dos problemas propostos e apresentados os resultados obtidos. Neste aspecto foram estabelecidas as seguintes métricas: tempo de execução de cada método e tempo total gasto da execução da tarefa. Para a coleta destes dados foi utilizado o *profiling*. Outra métrica que será analisada é a quantidade de linhas de código que se faz presente nos métodos analisados (convencional e lambda). A Seção 5 será destinado a descrever o passo a passo dos testes, utilizando a ferramenta mencionada. Os resultados obtidos foram apresentados e discutidos na Seção 6.

2 FUNDAMENTAÇÃO TEÓRICA

Antes do entendimento dos impactos da utilização da Expressão Lambda na programação orientada a objetos, é necessário abordar os paradigmas que os representam. Neste aspecto, paradigmas de programação buscam a resolução de problemas fornecendo uma percepção e especificando como o programador estrutura e executa um programa. Nas últimas décadas alguns paradigmas evoluíram, como o paradigma funcional e orientado a objetos. Algumas linguagens de programação suportam mais de um paradigma (TUCKER e NOONAN, 2010, p. 3).

2.1 PROGRAMAÇÃO FUNCIONAL

O paradigma de programação funcional modela um problema computacional através de funções matemáticas, ou seja, a saída de um programa depende única e exclusivamente de sua entrada (TUCKER e NOONAN, 2010, p. 4). Segundo Tucker e Noonan (2010, p. 361), o paradigma funcional surgiu em 1960 para subsidiar os pesquisadores da inteligência artificial e seus subcampos (computação simbólica, prova de teoremas, sistemas baseados em regras e processamento de linguagem natural).

Nesta época, a programação imperativa não dava um suporte adequado no tratamento dos problemas mencionados acima. Nela, temos a combinação de expressões que incluem valores concretos, variáveis e funções. Na PF, as funções podem ser utilizadas como argumentos, valores ou entradas para outras funções. Ela tem por base o **cálculo lambda**, desenvolvido por Alonzo Church em 1941, com o objetivo de modelar a classe das funções ”computáveis[2]”. Podemos definir o cálculo lambda a partir do formalismo abaixo.

Definição: Expressão Lambda

- Uma variável ou identificador é uma expressão lambda.
- Se A e B são expressões lambda, então (A B) é uma expressão lambda (aplicação).
- Se x é um identificador e B uma expressão lambda, então $(\lambda x\,.\,B)$, é uma expressão lambda (abstração).

[2] É a classe de problema solucionável ou parcialmente solucionável. Fonte: Diverio e Menezes, 1999, p. 167-168.

A alfa-equivalência é a definição para temos lambda equivalentes, seja a expressão lambda dada na forma de: $(\lambda x\,.\,x)$, é equivalente a: $(\lambda y\,.\,y)$, logo elas representam a mesma função identidade. Neste aspecto, as abstrações acima apresentam variáveis ligadas, e o nome da variável é igual ao parâmetro. Caso contrário, temos uma variável livre representada pela expressão que segue $(\lambda x\,.\,y)$, que é conhecida como função constante (ALLEN e MORONUKI, 2016, p. 35-37).

Sua aplicação se dá por meio de uma **redução beta**. Este processo simplifica uma expressão lambda de acordo com formalismo $\big((\lambda x\,.\,A)B\big) \Longrightarrow A[x \leftarrow B]$ (TUCKER e NOONAN, 2010, p. 364). Desta forma, temos que a substituição da aplicação B, é dada por todas as ocorrências livres de x em A por B. Ao fim de uma redução beta, esgotado todos métodos de redução, dizemos que uma abstração lambda se encontra em sua forma normal (ROSA, 2016, p. 16).

A **transparência referencial** é uma característica da PF. Para cada função executada ela irá produzir o mesmo resultado, desde que seus argumentos sejam os mesmos. Desta forma evitamos os **efeitos colaterais**, já que a execução de um programa não modifica qualquer variável externa. Exemplo: quando uma função altera o valor de uma variável global de uma classe, que está sendo utilizada por outra função (SILVA, 2015, p. 26).

Segundo Oliveira (2017, p. 26), quando uma linguagem de programação funcional não utiliza variáveis ela é denominada **pura**. Logo, as iterações desta linguagem são executadas por meio de recursão. Ainda conforme Oliveira (2017, p. 26), os programas são representados como funções e, quando executados, as especificações das funções serão avaliadas. Quando uma função utiliza outra função como argumento elas são nomeadas de **Funções de Alta Ordem.**

LISP (*LISt Processing*) foi a primeira linguagem funcional. Criada por John McCarthy em 1960, utilizada para o processamento simbólico de dados matemáticos. Como o passar dos anos outras linguagens foram surgindo como a Scheme, criada no MIT na década de 70. Enquanto que Haskell foi lançada no final dos anos 80 (TUCKER e NOONAN, 2010, p. 362-388). A utilização da programação funcional, em sistemas comerciais, não é amplamente difundida devido a dois motivos: dificuldade na aprendizagem do paradigma, em virtude da mudança de pensamento ao implementar a aplicação; e a perda de desempenho em comparação a programação imperativa, devido a arquitetura dos hardwares. Contudo, o interesse pelo paradigma vem crescendo, exemplo disso, é a incorporação dos recursos funcionais nas linguagens F#. A explicação para isso, deve-se a crescente utilização da programação concorrente, que utiliza múltiplos processadores, e a PF permitindo ao programador

desenvolver um sistema em um nível acima do imperativo, facilitando os testes e a manutenção do sistema tornando o mesmo mais robusto (HINSEM, 2009. apud ÁVILA, 2017, p. 29).

John Backus, na palestra que fez durante a cerimônia de entrega do Prémio Turing ACM de 1977, falou sobre as linguagens de programação puramente funcionais. Segundo ele, as linguagens funcionais puras são melhores do que as imperativas devido a sua legibilidade, por ser fidedigna, por apresentar menos erros e de fácil entendimento durante e depois do desenvolvimento. (JOHN BACKUS, 1978. apud SEBESTA, 2011, p. 682)

2.2 PROGRAMAÇÃO ORIENTADA A OBJETOS

O paradigma de Programação Orientada a Objeto (POO) foi o que mais evoluiu nos últimos anos. Atualmente é amplamente difundido e utilizado devido a sua robustez e eficiência, em segurança e reaproveitamento de código. As ideias iniciais da POO foram aplicadas na linguagem de programação "Simula 67". Só nos anos 80 que a POO foi tomando forma com o surgimento do "Smalltalk 80", que para muitos é considerada a base para a linguagem de programação puramente orientada a objeto (SEBESTA, 2011, p. 546).

"A POO fornece um modelo no qual um programa é uma coleção de objetos que interagem entre si, passando mensagens que transformam seu estado. Neste sentido, a passagem de mensagens permite que objetos de dados se tornem ativos em vez de passivos" (TUCKER e NOONAN, 2010, p. 4). Para um melhor entendimento deste paradigma é necessário abordar os seus conceitos.

2.2.1 Abstração

Uma abstração é um aspecto ou uma reprodução de uma entidade que apresenta os atributos mais significativos. Esta característica abstrai as complexidades de um programa ou rotina de um sistema, com isso o usuário não necessita saber dos detalhes da implementação do sistema, só o que ele faz, os resultados a partir de uma entrada (SEBESTA, 2011, p. 504). Vejamos: quando estamos dirigindo um veículo, não temos a necessidade de entender de mecânica. Sabemos que ao pisar no pedal de freio o carro para, "escondendo" do motorista os mecanismos para realizar esta ação, abstraindo toda complexidade do sistema (DEITEL e DEITEL, 2010, p. 57). Segundo Sebesta (2011, p. 504), existem dois tipos de abstração nas linguagens atuais: as abstrações de processos e abstração de dados. O conceito de abstração de

processo está associado aos subprogramas de um programa. Quando é necessário realizar um processo o programa vai realizá-lo sem mostrar os detalhes de como isso é feito. Este procedimento é feito por subprogramas. Já na abstração de dados, temos a representação do encapsulamento de atributos e subprogramas que darão características e funções a um objeto. Sendo ele um exemplo da abstração de dados.

2.2.2 Encapsulamento

O encapsulamento é um dos pilares da programação orientada a objeto. Esta técnica oculta as informações ou dados do objeto. Para acessar as informações dele é necessário utilizar os métodos de acesso chamados: *Getters* e *Setters* (GASPAROTTO, 2014, on-line).

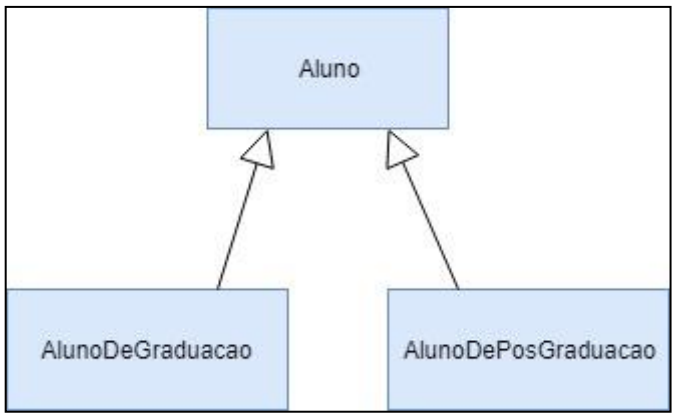
Para poder ocultar as informações do objeto é necessária a utilização do modificador de acesso *private.* Sua utilização é corriqueira em variáveis de instâncias e os dados só serão visíveis dentro do programa corrente. Existem outros dispositivos de acesso, como o *public,* em que as informações serão visíveis a qualquer classe ou método. Podemos destacar ainda o *protected*, seus dados serão visíveis para um método do programa. Este modificador é bastante utilizado quando utilizamos herança (TUCKER e NOONAN, 2010, p. 318).

O acesso às informações do objeto fica por conta dos métodos *getters* e *setters.* Eles apresentam o modificador de acesso *public,* desta maneira os clientes podem configurá-los. Os métodos de acesso *getters* obtém os valores das variáveis de instâncias e os *setters* alteram estes valores (DEITEL e DEITEL, 2010, p. 66).

2.2.3 Herança

Para Deitel e Deitel (2017, p. 284-285), herança é uma maneira de utilizar códigos já existentes a partir de uma classe criada. Uma classe-filha ou subclasse herda os atributos e métodos de uma classe-mãe ou superclasse. Neste sentido, as superclasses tendem a ser mais gerais e as subclasses mais específicas. As relações de herança apresentam uma estrutura em árvore, hierárquicas. Imagine uma universidade... ela é composta por alunos que podem ser de graduação ou pós-graduação (Figura 1). Na Figura 1, as subclasses *AlunoDeGraduacao()* e *AlunoDePosGraduacao()* herdam as especificações (atributos e métodos) da superclasse *Aluno*.

Figura 1 - Hierarquia de herança simples.



Fonte: Adaptada. Deitel e Deitel, 2010.

Na POO temos duas variações de herança: *simples e múltipla.* Na herança simples, uma subclasse B herda o estado e o comportamento de uma superclasse A. Na herança múltipla, uma subclasse D herda o estado e o comportamento de uma ou mais superclasses. Linguagens de programação como C++, Python e Eiffel suportam herança múltipla, já as linguagens Java e Smalltalk suportam somente herança simples (TUCKER e NOONAN, 2010, p. 319-323).

2.2.4 Polimorfismo

Outro conceito muito utilizado na programação orientada a objetos é o polimorfismo. Ele permite que uma ou mais classes herdem as mesmas características de uma superclasse. Contudo, as implementações das subclasses são diferentes, seu funcionamento interno é alterado. Vejamos.... Considere o seguinte programa: Queremos simular a emissão de sons de alguns animais, com isso temos as classes *Cachorro, Gato e Pássaro.* Essas subclasses herdam as características da superclasse. *Animal,* que apresenta o método *emitirSom().* Perceba que quando o método for acionado pelas subclasses, seu resultado virá de formas diferentes, já que um cachorro não emite som igual a um gato ou um pássaro. Ou seja, temos a mesma mensagem para resultados diferentes (DEITEL e DEITEL, 2017, p. 312-313).

2.2.5 Método e Atributos

Para um programa realizar um procedimento é necessário um método, ele esconde as informações de como irá realizar um procedimento, assim como um botão de ligar ou desligar oculta as tarefas necessárias para energizar um equipamento eletrônico (televisão, som, máquina de lavar). O seu acionamento para realização de tarefas é por meio de mensagens ou chamada de método, em sua maioria elas são feitas por um objeto. Quando um método é invocado por um objeto, ele é conhecido como método de instância. Existem os métodos *static* ou métodos de classe que não é necessária a criação de um objeto (DEITEL e DEITEL, 2010, p. 57; TUCKER e NOONAN, 2010, p. 317-318).

Os métodos podem apresentar parâmetros ou argumentos. Eles são informações necessárias para a execução de uma ação, podendo ser de vários tipos. Ao final de sua execução, os métodos podem ou não retornar algum valor, eles só suportam um único tipo de retorno.

As variáveis de instância, declaradas dentro de uma classe, são conhecidas como atributos de um objeto que definem as características do mesmo. Um atributo deve apresentar um nome e ser de um tipo. Cada objeto possui seu atributo, ou seja, dado uma classe do tipo *Pessoa()* que contém os atributos nome e idade, quando instanciada, cada objeto terá seus próprios atributos e poderão receber valores diferentes (DEITEL e DEITEL, 2017, p. 58-73).

2.2.6 Classes e Objetos

Uma classe é uma estrutura que apresenta abstração de dados ocultando as informações e funções que caracterizam um objeto. As funções de uma classe são representadas pelos *métodos* e o objeto é a representação da instância de uma classe. Na POO as classes realizam duas tarefas complementares importantes: o tipo do objeto é determinado por elas e permitem a verificação total de tipos (TUCKER e NOONAN, 2010, p. 315-318).

Na POO podemos definir classes abstratas. Isso acontece quando um ou mais métodos são abstratos. Um método é abstrato quando apresenta somente uma assinatura, ou seja, não apresenta código em seu escopo. Classes abstratas são utilizadas como superclasses, desta forma outras classes podem herdar seu padrão. Elas não podem ser instanciadas pois são consideradas incompletas. Isso porque existem métodos que precisam ser implementados já que só as classes concretas são instanciadas (DEITEL e DEITEL, 2017, p. 316-318).

As classes podem apresentar dois tipos de métodos. O método de instância, trabalha apenas nos objetos da classe. Métodos da classe executam trabalhos dentro da classe e nos objetos da classe. Elas podem apresentar ainda dois tipos de variáveis. Variáveis de instância são as representações dos atributos de um objeto podendo ser manipuladas por qualquer método da classe. E as variáveis de classe são as variáveis estáticas: elas pertencem à classe e só vamos ter uma cópia por classe (SEBESTA, 2011, p. 548).

O objeto é a representação de uma classe. Vejamos, antes de construir uma casa é necessária a criação de um projeto. Ele contém as plantas e as instruções necessárias para fazê-la. As plantas vão conter por exemplo, dimensões de paredes e cômodos, entradas e saídas, localização da instalação hidráulica e elétrica, entre outras. A partir destas plantas e do projeto é que se constroem as casas. Fazendo uma analogia à POO, as plantas representam as classes e as casas os objetos. A partir de uma planta podemos construir várias casas, assim como a partir de uma classe concreta podemos instanciar vários objetos (DEITEL e DEITEL, 2010, 58-60).

2.2.7 Vantagem e Desvantagem

Para Biondo (2017, p. 23) e Gasparotto (2014) a programação orientada a objetos apresenta reuso de código, sendo esta uma das principais vantagens na produção de um sistema. A reutilização de código, ou de módulos, vai minimizar o tempo de produção deste sistema. Como a POO faz um paralelo com o mundo real, o entendimento e a legibilidade do paradigma ficam mais simples de assimilar. Desta forma, a manutenção do sistema poderá ser feita por mais de uma pessoa.

Outro aspecto importante é a modularidade. Com a POO podemos dividir um sistema de difícil compreensão em partes menores, simplificando o problema, facilitando sua solução, manutenção e evolução do sistema (PINHO, 2012, p. 8). Contudo, um sistema orientado a objeto tende a ser mais lento ao dividir o problema em blocos, devido a sua comunicação que é por troca de mensagens. Entretanto, há quem diga que essa lentidão não é percebida devido as configurações dos hardwares atuais, com o aumento do processamento e maior quantidade de memória. Neste sentido, a perda de desempenho do sistema pode estar associada a erros na programação ou a forma como o código foi escrito (GASPAROTTO, 2014).

2.3 A LINGUAGEM JAVA

Antes de se tornar a linguagem mais utilizada no mundo, segundo Tiobe (2019, on-line), o Java era utilizado para programação de televisão digital a cabo. Sua história começa nos anos 90, quando ainda se chamava Oak, através das mãos de James Gosling ela foi criada, na época ele era chefe do projeto Green. Devido a sua robustez e poder, a linguagem evoluiu e expandiu seus horizontes sendo aproveitada na Internet. O seu nome foi modificado devido a problemas no registro, passando a se chamar Java. Foi em 1995 que a Sun percebeu que poderia utilizá-la em ambiente *web*. Nesta época não havia conteúdo dinâmico, interatividade e animações (TAVARES e CALDAS, 2017, p. 13). Devido ao sucesso ela chamou a atenção dos empresários e começou a ser utilizada em sistemas comerciais de grande porte, servidores da *web*, telefones celulares, *smartphones*, entre outros (DEITEL e DEITEL, 2017, p. 13).

Com a evolução da Internet e da *World Wide Web* (WWW), a equipe de desenvolvedores formulou um documento contendo princípios da linguagem para participar mais ativamente do mercado virtual e *e-Commerce*. Nele temos os objetivos da linguagem e a descrição de como ela deve se comportar (TAVARES e CALDAS, 2017, p. 14).

2.3.1 Simples, orientado a objeto e familiar

Desenvolvida no paradigma orientado a objeto, a linguagem foca na simplicidade apresentando um ambiente no qual os programadores não precisam realizar estudos exaustivos para o entendimento dos seus conceitos. Não é coincidência que ela apresenta uma aparência e funcionalidade semelhante a linguagem C++. Isso significa que os programadores podem migrar para o Java facilmente (familiaridade).

2.3.2 Robusto e Seguro

Desenvolvida para criar sistemas seguros e confiáveis, a linguagem apresenta verificações em tempo de compilação e de execução, não bastando isso, fornecem aos desenvolvedores mecanismos para hábitos de programação seguros.

2.3.3 Arquitetura Neutra e Portátil

Uma das principais características da linguagem é a sua portabilidade pois ela independe de arquitetura. Neste aspecto um aplicativo pode ser executado em qualquer sistema operacional, para isso, no entanto, é necessário ter instalado a JVM (*Java Virtual Machine*). Ao realizar a compilação do código Java geram-se os *bytecodes* os quais são direcionados a máquina virtual (JVM) que irá interpretá-los para os diversos sistemas operacionais.

2.3.4 Alto Desempenho

A JVM fornece uma gama de recursos para o idioma. *Garbage Collector*, é uma função que limpa a memória virtual de forma automática. Desta forma, o ambiente está sempre pronto para garantir recursos para o sistema fornecendo um melhor desempenho quando necessário.

2.3.5 Interpretada, Dinâmica e Encadeada

A JVM é responsável pela a interpretação dos *bytecodes.* Ela irá se comunicar com o sistema operacional, proporcionando um desenvolvimento rápido e em ciclo. A dinâmica da linguagem pode ser percebida quando a ligação de classes é realizada em tempo real e só quando necessário. Com a utilização da classe *Thread* a linguagem possibilita a utilização do paralelismo.

2.4 JAVA SE 8

A contínua evolução é uma marca da linguagem Java. Em 2014 ela disponibilizou a versão 8. Neste sentido, o Java SE 8 apresentou modificações em termos de linguagem, segurança, coleções, ferramentas e bibliotecas (TAVARES e CALDAS, 2017, p. 15). Algumas modificações são mais significativas, elas modificam como os programadores interagem com a codificação do programa, estas melhorias podem ser encontradas em (ORACLE, 2019, on-line). Dentre elas estão presentes as Expressões Lambda. Este novo recurso aumenta a força da linguagem pois temos a adição de características do Paradigma Funcional; a descrição deste recurso está presente no Projeto Lambda[3] (DEITEL e DEITEL, 2017, p. 574). A partir de agora

[3] http://openjdk.java.net/projects/lambda/

nos debruçaremos sobre este novo conceito presente no Java SE 8, observando os conceitos, características, sintaxe e exemplos.

2.4.1 Expressões Lambda Java SE 8

Conforme já mencionado, a equipe da Oracle, empresa que mantém o Java, apresentou uma nova abordagem para tratar o problema das Classes Internas Anônimas. Quando utilizamos este recurso, geralmente temos grande quantidade de código sem utilidade real. Buscando melhorias neste aspecto a linguagem adicionou algumas características utilizadas na PF, modificando a forma do programa e diminuindo a quantidade de código exigida para executar uma tarefa. Expressões Lambda foi o nome selecionado para a funcionalidade. No entanto, *closures* ou *anonymous methods* (CARVALHO, 2015, p.18-19) são outras maneiras de se referir a ela. Para utilizar este recurso no Java SE 8 é necessária uma interface funcional, falaremos dela mais adiante.

Para nosso primeiro exemplo, temos um evento sobre um botão. A Listagem 1 apresenta o evento em termos de Classes Internas Anônimas. Em seguida vamos reescrever na Listagem 2 utilizando as Expressões Lambda.

Listagem 1 - Utilizando Classe Anônima.

```
1  import java.applet.Applet;
2  import java.awt.*;
3  import java.awt.event.ActionEvent;
4  import java.awt.event.ActionListener;
5  class AWT1Exemplo extends Frame {
6
7      private Button botao;
8
9      public AWT1Exemplo() {
10         botao = new Button("Botão");
11         add(botao);
12
13         // Forma convencional, Classe Interna Anônima
14         botao.addActionListener(new ActionListener() {
15             public void actionPerformed(ActionEvent ae) {
16                 System.out.println("Clicou")
17                 }
18         });
19     }
20 }
```

Fonte: Carvalho, 2015.

Listagem 2 - Utilizando Expressões Lambda

```
1  class Swing1Exemplo extends Frame {
2      private JButton botao;
3
4       public Swing1Exemplo () {
5          botao = new JButton("Botão");
6          add(botao);
7
8      // Utilizando ExpressÃµes Lambda
9      botao.addActionListener(e-> System.out.println("Clicou"));
10     }
11 }
```

Fonte: Carvalho, 2015.

Nas listagens podemos destacar a quantidade de linhas de código que diminuíram. Para executar uma ação no botão são necessárias cinco linhas de código, da linha 14 (quatorze) a 18 (dezoito), no primeiro exemplo. Já no segundo exemplo temos apenas uma linha, a 9 (nove) (CARVALHO, 2015, p. 19-20).

Diferente da programação imperativa que demonstra como fazer uma tarefa, na programação funcional o programador declara o que quer alcançar para executar um programa. Não é necessário, por exemplo, declarar variáveis mutáveis ou iterar sobre elementos. Isso porque a biblioteca do Java SE 8 determina como acessar todos os elementos, isso é conhecido com **repetição interna.** Com esta característica podemos realizar processamento paralelo, aproveitando ao máximo a arquitetura multiprocessada do computador (DEITEL e DEITEL, 2017, p. 572-573).

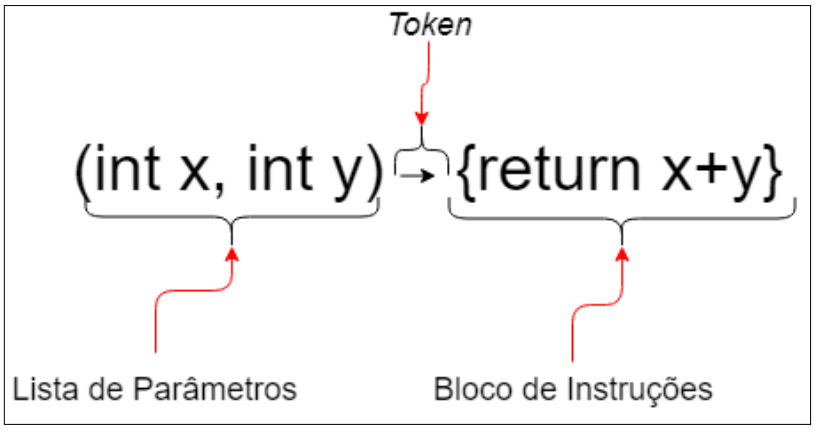
2.4.2 Sintaxe

Sua representação é expressa da seguinte forma (TAVARES e CALDAS, 2017, p. 18):

- Uma lista de parâmetros separados por vírgulas dentro de parênteses;
- Um *token*, uma seta;
- Um corpo podendo ser uma única expressão ou um bloco de instruções.

Vejamos na Figura 2:

Figura 2 - Representação da sintaxe de uma Expressão Lambda Java SE 8.



Fonte: Adaptada. Deitel e Deitel,2017.

A sintaxe pode apresentar algumas variações (DEITEL e DEITEL, 2017, p. 574-575). Vejamos:

Quando temos mais de um parâmetro e seus tipos são iguais podemos omití-los. Neste sentido o compilador é quem determina o tipo do parâmetro.

- $(x, y) \longrightarrow \{return\ x + y\}$

Quando temos uma única instrução em seu corpo, podemos omitir a palavra-chave *return* e as chaves, seu retorno será de forma implícita. Vejamos:

- $(x, y) \longrightarrow x + y$

Se tivemos um único parâmetro, podemos omitir os parênteses. Vejamos:

- $value \longrightarrow System.out.printf("\%d", value)$

Podemos definir lambda com parâmetro vazio, é só colocar lista de parâmetros entre parênteses vazio à esquerda do *token*, (seta). Vejamos:

- $() \longrightarrow System.out.printf("Bem\ vindos,\ lambda!!!")$

2.4.3 Utilizando expressões lambda

Para podermos utilizar este recurso temos que ter uma interface funcional. É uma interface que contém um único método abstrato. Neste sentido, utilizaremos as Expressões Lambda para sobrescrever o comportamento do método abstrato. Esta interface pode conter outros métodos como *default* e *static*. Para termos certeza que estamos utilizando uma interface funcional, foi criada uma nova anotação adicionada no Java SE 8, *@FunctionalInterface.* Ao adicionar esta descrição em uma interface o compilador verifica se ela satisfaz os requisitos do seu modelo (TAVARES e CALDAS, 2017, p. 19). O pacote *java.util.function* contém várias interfaces funcionais utilizadas no Java. No Quadro 1 segue uma relação mostrando as principais interfaces deste pacote. T e R são nomes genéricos que representam os tipos de objetos e retorno de um método (DEITEL e DEITEL, 2017, p. 574).

Quadro 1 - Interfaces funcionais básicas do pacote (java.util.function).

Interface	Descrição
BinaryOperator<T>	Contém o método *apply*, que recebe dois argumentos T, realiza uma operação neles (como um cálculo) e retorna um valor do tipo T.
Consumer<T>	Contém o método *accept*, que recebe um argumento T e retorna void. Realiza uma tarefa com o argumento T, como gerar uma saída do objeto.
Function<T, R>	Contém o método *apply*, que recebe um argumento T e retorna um valor do tipo R.
Predicate<T>	Contém o método *test*, que recebe um argumento T e retorna um boolean.
Supplier<T>	Contém o método *get*, que não recebe argumentos e produz um valor do tipo T
UnaryOperator<T>	Contém o método *get* que não recebe argumentos e retorna um valor do tipo T.

Fonte: Adaptada. Deitel e Deitel, 2017.

A utilização das expressões lambda não se resume apenas a essas interfaces pois, de acordo com a definição, uma interface que contém um método abstrato pode utilizar as expressões lambda. Desta forma, podemos utilizar as interfaces antigas, por exemplo a *Runnable*. O lambda no Java SE 8 só existem se associarmos com as interfaces funcionais. A não ocorrência disso gera um erro de compilação (CARVALHO, 2015, p . 21). A seguir

veremos como isso funciona. Temos então uma interface funcional. Listagem 3 (***Validador<T>)*** que apresenta o método ***validar (T t)*** e retorna um ***boolean****:*

Listagem 3 - Interface Funcional Validador

```
1  package interfacesFuncionaisCriandoElas;
2
3  @FunctionalInterface
4  public interface Validador<T> {
5
6      boolean validar(T t);
7  }
8
```

Fonte: Silveira e Turini, 2014.

A Listagem 4 apresenta a classe principal que dispõe do método *main* que testa o método ***validar (T t)***. Neste primeiro momento vamos utilizar a forma convencional, com a utilização de classes anônimas. Em seguida, na Listagem 5, temos a utilização da Expressão Lambda como o novo recurso a partir do Java 8.

Listagem 4 - Classe Principal, utilizando a interface Validador com Classe Anônima.

```
1  package interfacesFuncionaisCriandoElas;
2
3  public class Principal {
4
5      public static void main(String[] args) {
6
7          // Forma convencional, utilizando classe anônima
8          Validador<String> validadorCEP = new Validador<String>() {
9              public boolean validar(String valor) {
10                 return valor.matches("[0-9]{5}-[0-9]{3}");
11             }
12         };
13
14         boolean b = validadorCEP.validar("04101-300");
15         System.out.println(b);
16
17     }
18
19 }
20
```

Fonte: Silveira e Turini, 2014.

Como estamos utilizando uma interface funcional, podemos utilizar as Expressões Lambda. Vejamos na Listagem 5:

Listagem 5 - Classe Principal, utilizando a interface Validador com Expressão Lambda.

```
1  package interfacesFuncionaisCriandoElas;
2
3  public class Principal {
4
5      public static void main(String[] args) {
6
7          // Novo formato, utilizando expressões lambdas.
8          Validador<String> validadorCEP = valor -> valor.matches("[0-9]{5}-[0-9]{3}");
9
10         boolean b = validadorCEP.validar("04101-300");
11         System.out.println(b);
12
13     }
14
15 }
16
```

Fonte: Silveira e Turini, 2014.

No exemplo da Listagem 5, criamos nossa própria interface mas, como já foi dito, o Java possui várias interfaces funcionais disponíveis em suas bibliotecas. Com a utilização do lambda podemos destacar a legibilidade, melhorando o entendimento da função (SILVEIRA e TURINI, 2014, p. 14-15).

Com atualização da API temos adição do pacote ***Stream*** (*java.util.Stream*). Este novo recurso permite a manipulação de coleções no Java a partir de sua versão 8 de forma simplificada. Os objetos das classes que implementam a interface ***Stream*** são conhecidos como fluxos. Neste sentido podemos criar *Pipelines* de fluxo que são sequências de processamento em um elemento, iniciando com uma coleção ou *array*, realizando operações intermediárias e concluindo com operação terminal. Desta maneira temos um encadeamento de método e o fluxo não mantém as informações dos elementos, sendo assim ele não pode ser reutilizado. Como já dito, os fluxos realizam operações **terminais** e **intermediárias.** As operações terminais são conhecidas como “gulosas” – quando são requisitadas, realizam as operações de forma completa e iniciam o processamento das operações intermediárias. O Quadro 2 lista algumas operações terminais utilizadas. Já as operações intermediárias são conhecidas como “preguiçosas” – imagine um *array* de inteiros que está desordenado, procuramos então o numeral 5 (cinco), se ele for o primeiro elemento não é necessário percorrer toda lista. Neste sentido ela realiza operações sobre o elemento resultando em um novo fluxo. O Quadro 3 apresenta as operações intermediárias (DEITEL e DEITEL, 2017, p. 575-576).

Quadro 2 - Operações Stream terminais

Operações Stream terminais	
forEach	Em um fluxo o processamento é realizado em cada elemento.
Average	A média dos elementos de um fluxo numérico é calculada.
Reduce	Utilizando uma função que acumula valores, reduz elementos de uma coleção.
Collect	Com os resultados anteriores de um fluxo é criada uma nova coleção de elementos.
findFirst	Localiza o primeiro elemento de fluxo em uma coleção do mesmo.

Fonte: Adaptada. Deitel e Deitel, (2017)

Quadro 3 - Operações Stream intermediárias

Operações Stream intermediárias	
filter	Novo fluxo, contendo só elementos que atendem a uma condição.
Distinct	Novo fluxo, que contém só elementos únicos.
limit	Novo fluxo, contendo um número específico de elementos a partir da origem do fluxo original.
map	Novo fluxo, contendo o mesmo número de elemento da coleção original. Aqui cada elemento do fluxo original será mapeado para um novo valor.
sorted	Novo fluxo, que contém elementos ordenados.

Fonte: Adaptada. Deitel e Deitel, (2017)

Vejamos então com isso ocorre. O ***Stream*** realiza operações em *arrays* e coleções. No exemplo que segue temos uma coleção de objeto do tipo Empregado, queremos então saber os nomes dos empregados que recebem um salário maior que R$ 2.000. Uma das maneiras de representar esta pesquisa, de forma convencional, é apresentada na Listagem 6. Já na Listagem 7 reescrevemos utilizando as Expressões Lambda.

Listagem 6 - Sem Stream, forma convencional.

```
3  import java.util.Arrays;
4  import java.util.List;
5
6  public class Teste2 {
7
8      public static void main(String[] args) {
9          Empregado empregado1 = new Empregado("Carlos Alberto", 1500.00);
10         Empregado empregado2 = new Empregado("Maria Aparecida", 3000.00);
11         Empregado empregado3 = new Empregado("Antonio Jesus", 2000.00);
12         Empregado empregado4 = new Empregado("Marlene Santos", 2500.00);
13
14         List<Empregado> empregados = Arrays.asList(empregado1, empregado2, empregado3, empregado4);
15
16         for (Empregado empregado : empregados) {
17             if (empregado.getSalario() > 2000) {
18                 System.out.println(empregado.getNome());
19             }
20         }
21     }
22 }
```

Fonte: Adaptada, Silveira e Turini, 2014.

A Listagem 6 apresenta a forma convencional no qual é utilizado o comando ***for*** (linha 16) que vai iterar sobre a coleção de forma explícita. Temos também o comando ***if*** (linha 17) para filtrar os nomes dos empregados que recebem salário maior que R$ 2.000.

Listagem 7 - Com Stream, utilizando Lambda

```
3  import java.util.Arrays;
4  import java.util.List;
5
6  public class Teste2 {
7
8      public static void main(String[] args) {
9          Empregado empregado1 = new Empregado("Carlos Alberto", 1500.00);
10         Empregado empregado2 = new Empregado("Maria Aparecida", 3000.00);
11         Empregado empregado3 = new Empregado("Antonio Jesus", 2000.00);
12         Empregado empregado4 = new Empregado("Marlene Santos", 2500.00);
13
14         List<Empregado> empregados = Arrays.asList(empregado1, empregado2, empregado3, empregado4);
15
16         empregados.stream()
17             .filter(e -> e.getSalario() > 2000)
18             .forEach(e -> System.out.println(e.getNome()));
19     }
20 }
```

Fonte: Adaptada, Silveira e Turini, 2014.

Na Listagem 7 temos a utilização do ***Stream***. Perceba que não foi necessária a utilização de um laço para iterar sobre a coleção, ela é realizada internamente. Como já foi apresentado, o *filter* é uma operação intermediária, retorna um novo fluxo com elementos que atendem a uma condição. Para apresentar o resultado da pesquisa encadeamos o método terminal *forEach.* Podemos então perceber que sua utilização traz vantagens em termos de organização e entendimento do código, mas não é só isso, ***Stream*** não possui efeitos colaterais e os métodos

de sua interface não alteraram os elementos do fluxo original (SILVEIRA e TURINI, 2014, p. 42-44).

As expressões lambda, em alguns momentos, são utilizadas para chamar um método existente. Neste caso podemos referenciar o método ficando, assim, mais claro. (ORACLE, 2019, p. on-line). *O Method Reference,* foi adicionado no Java SE 8. Para sua utilização é necessária uma interface funcional, ou seja, em alguns casos podemos substituir uma expressão lambda por uma referência ao método. Quando referenciado, o método deve possuir o mesmo tipo de retorno e argumento de uma interface funcional (TEIXEIRA JUNIOR, 2014, p. 28). Existem quatro formas de utilização do recurso, apresentados no Quadro 4:

Quadro 4 - Tipos de Method Reference.

Lambda	Descrição
String::toUpperCase	Referência a um método de instância de um objeto arbitrário de um tipo específico.
System.out::println	Referência a um método de instância de um objeto específico
Math::sqrt	Referência a um método estático
TreeMap::new	Referência a um construtor

Fonte: Adaptada. Deitel e Deitel, 2017; Oracle, 2019.

2.5 PROFILING

O desempenho do sistema é um dos requisitos mais importantes para uma aplicação, mesmo ele não sendo um requisito funcional, o tempo de execução de *software*, para alguns domínios, é fundamental. Não é aceitável um sistema demorar para realizar tarefas comuns do dia a dia. Independente do ambiente o software deve apresentar tempo de resposta aceitável. Para os usuários, o tempo gasto por um aplicativo para realizar uma tarefa é uma qualidade do equipamento. Neste aspecto, umas das formas de verificar o seu desempenho é utilizando os *Profiling* (AMORIM, 2014, on-line).

O *profiling* é uma ferramenta que possibilita localizar erros de programação em tempo de execução. Neste aspecto ela busca medir o tempo de execução de um método, quantidade de vezes que ele foi invocado, quantidade de memória utilizada por ele e por seus submétodos, quantidade de memória alocada por objeto em tempo de execução, e o número de vezes que o *Garbage Collector (GC)* foi utilizado (NOGUEIRA, 2014, p. 5-6).

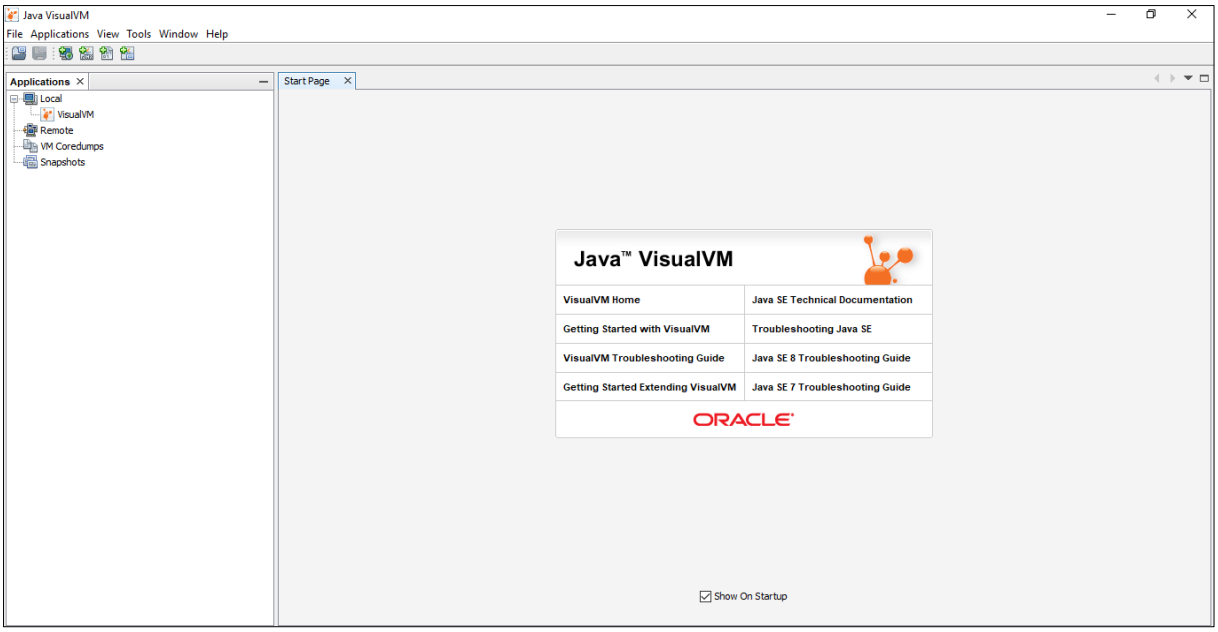
Existem vários *Profiling* entre pagos e *open-source*, cada um com suas características. Segundo Medeiros (2015), os mais utilizados são: J*Probe Suite* (*Quest Software*), *OptimizeIt Suite* (*Borland*) e o *JProfiler* (*Ej-Technologies*).

- O *JProbe Suite* é formado por quatro ferramentas: *JProbe Memory Debugger, JProbe Profiler, JProbe* Threadalyzer e *JProbe Coverage.* Cada uma possui um papel diferente, composta por dois componentes: o *JProbe Console* e o *JProbe Analysis*. O *JProbe Console*, fornece informações visuais coletadas através do *JProbe Analysis*.
- *OptmizeIt* verifica a utilização e alocação de memória e o uso ineficiente da *CPU*, analisando a *JVM*.
- *JProfiler* é uma ferramenta simples, porém poderosa, possibilitando várias análises em uma aplicação; entre elas temos a base de dados como: *JDBC, JPA/Hibernate, MongoDB* entre outras. Ferramenta com licença comercial.

Podemos destacar a utilização de outras ferramentas, como a *VisualVM*, ferramenta *open-source* que está presente no *JDK* a partir do Java 6. Permite análise local e remota, enquanto a aplicação está em execução na *JVM*, fornecendo informações visuais sobre o aplicativo. A ferramenta permite aos desenvolvedores gerar e analisar despejos de *heap*, rastrear vazamentos de memória, executar e monitorar a coleta de lixo e executar perfis leves de memória e *CPU*.

Para iniciar o *VisualVM* no Windows, vá até o diretório C: \ Arquivos de programas \ Java \ jdk 1.8.0_221 \ bin, dentro da pasta \bin, procure por *jvisualvm.exe*. Ao abrir o aplicativo teremos a tela vista na Figura 3.

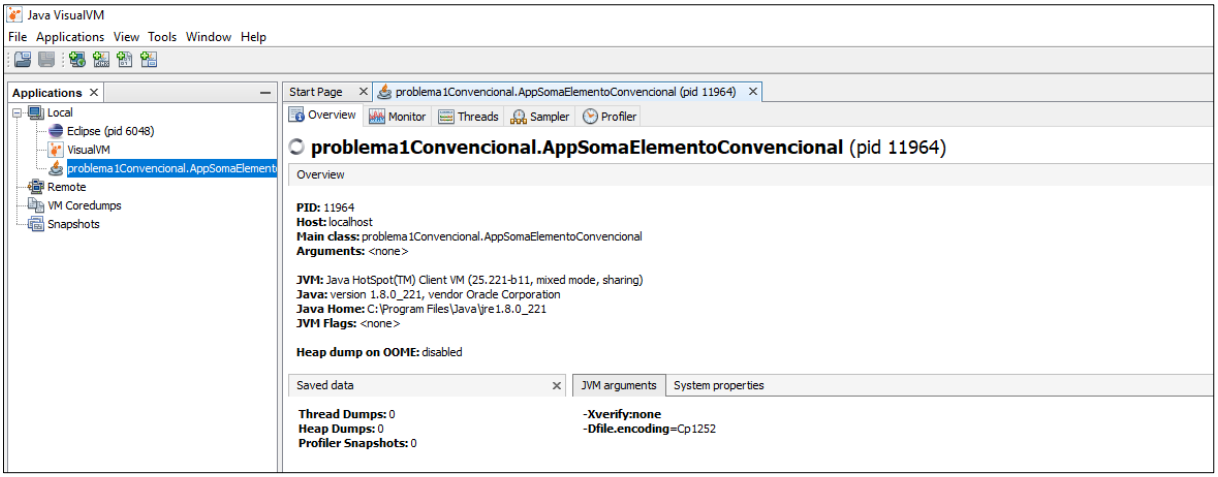
Figura 3 - *VisualVM*, tela inicial.



Fonte: Oracle (2019).

A janela é dividida em duas partes, *applications* à esquerda e *StartPage* à direita. O painel *applications* é o principal ponto de entrada para explorar os aplicativos em execução, com uma estrutura em árvore ela permite uma visualização mais rápida dos aplicativos. No painel *StartPage* serão exibidas as informações dos aplicativos, cada um é representado por uma aba.

Figura 4 -*VisualVM*, aba overview

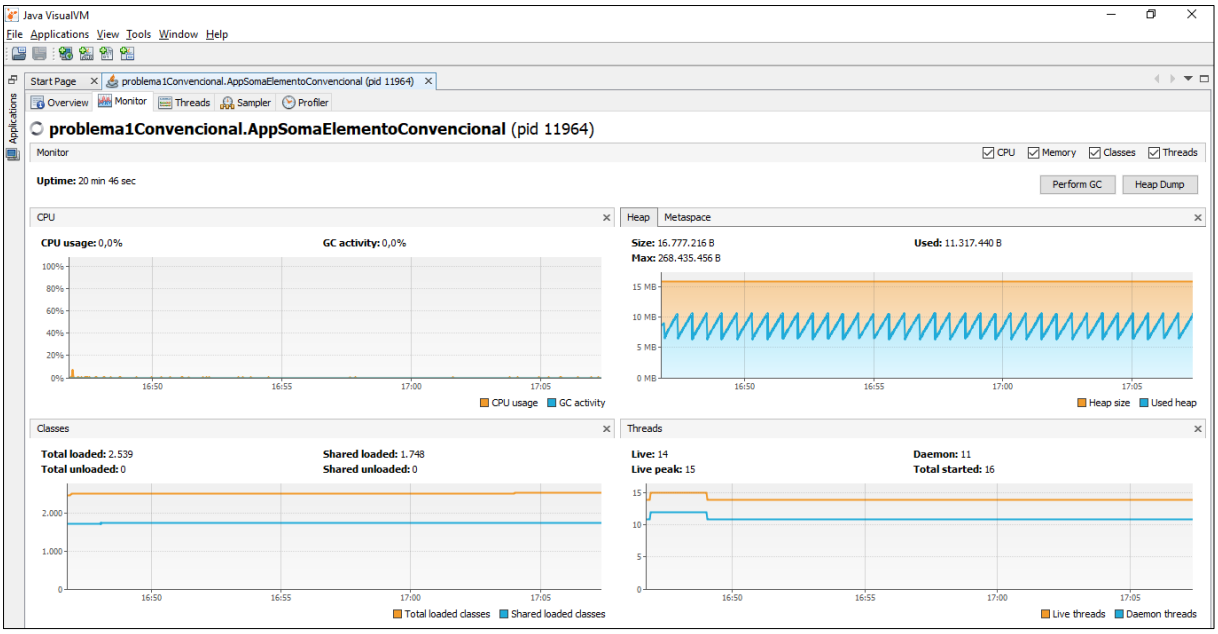


Fonte: Adaptada, Oracle (2019)

Ao inicializar uma aplicação a *JVM* a identifica de imediato. Clicando duas vezes temos na tela exatamente o que mostra a Figura 4. No painel *StartPage* são apresentadas algumas abas

que iremos descrever ao longo deste tópico. A aba *Overview*[4], é uma página destinada à informação, como o número do processo, a classe principal de execução da aplicação, versão da JVM e seus argumentos, conforme é ilustrado na Figura 4. Na aba monitor[5] (Figura 5), podemos visualizar dados gráficos de alto nível, ele representa a utilização da CPU, memória, classes utilizadas e *Threads* em execução. Ainda a partir desta aba, podemos invocar um *"Full Garbage Collection"* através do botão "*Perform GC*" e também podemos fazer um *"Heap Dump"*, assim como gravar dados instantâneos em arquivo "*hprof*" (*snapshot)*, contendo informações que haviam na *heap* naquele momento.

Figura 5 – VisualVM, aba Monitor.

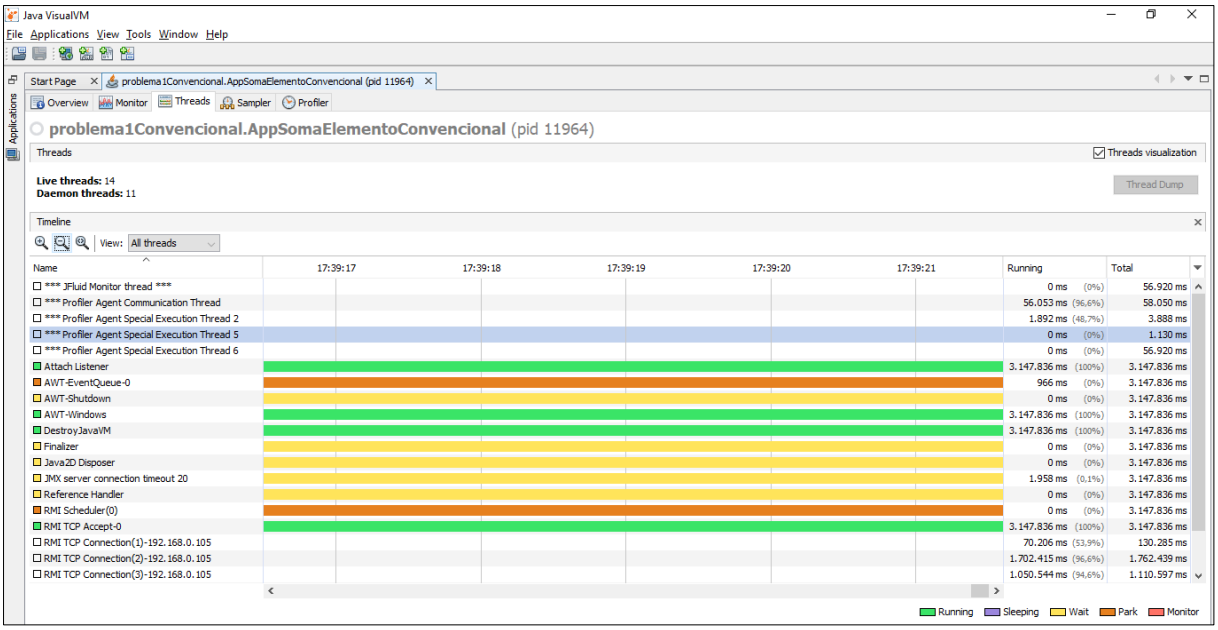


Fonte: Adaptada, Oracle (2019)

Na aba *Threads*[6], podemos visualizar o seu estado e em que momento ficaram em execução. Todo monitoramento é em tempo de execução, vejamos na Figura 6:

[4] VisualVM – Overview: https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/overview_tab.html
[5] VisualVm – Aba Monitor: https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/monitor_tab.html
[6] VisualVm – Aba Threads: https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/threads.html

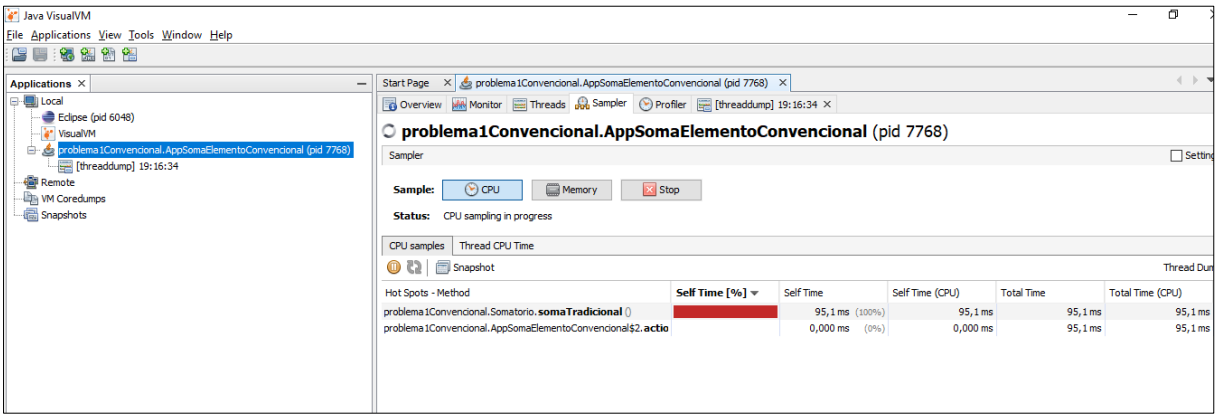
Figura 6 - VisualVM, aba *Threads*



Fonte: Adaptada, Oracle (2017)

Na aba *Sampler* (Figura 7), vamos poder analisar o desempenho da aplicação em relação ao uso da *CPU* e Memória. Com a aplicação ativa, clique no botão *CPU*, depois inicie o processo que a aplicação realizar. Este procedimento é válido para o botão memória. Durante o procedimento será exposta uma tabela contendo informações sobre o tempo de execução do método da aplicação.

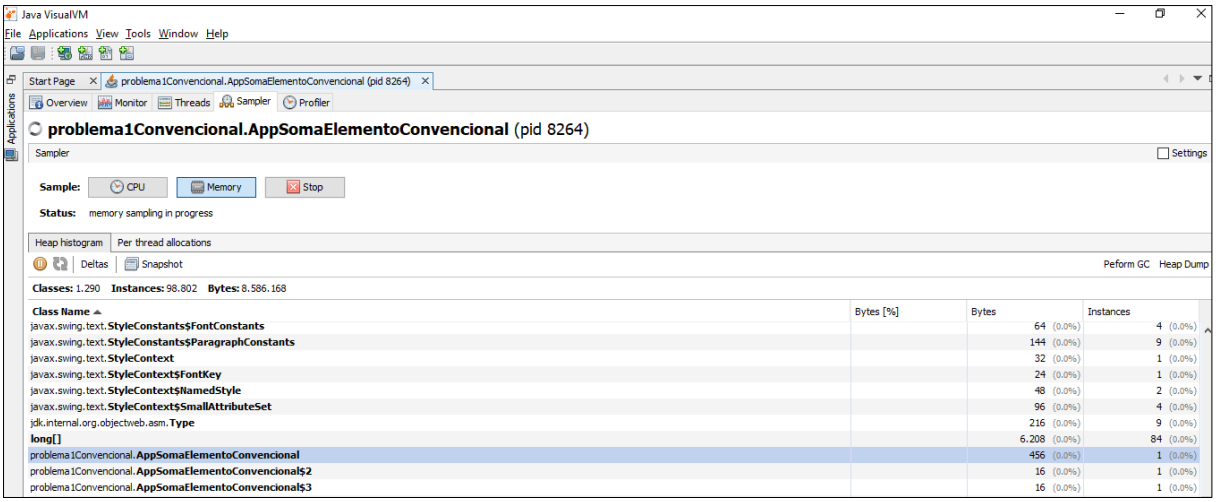
Figura 7 - VisualVM, *Sampler* CPU.



Fonte: Adaptada, Oracle (2019)

Para a memória teremos uma amostra, descrita em tabela, que irá apresentar a quantidade de *bytes* associada à classe e o número de instâncias (Figura 8).

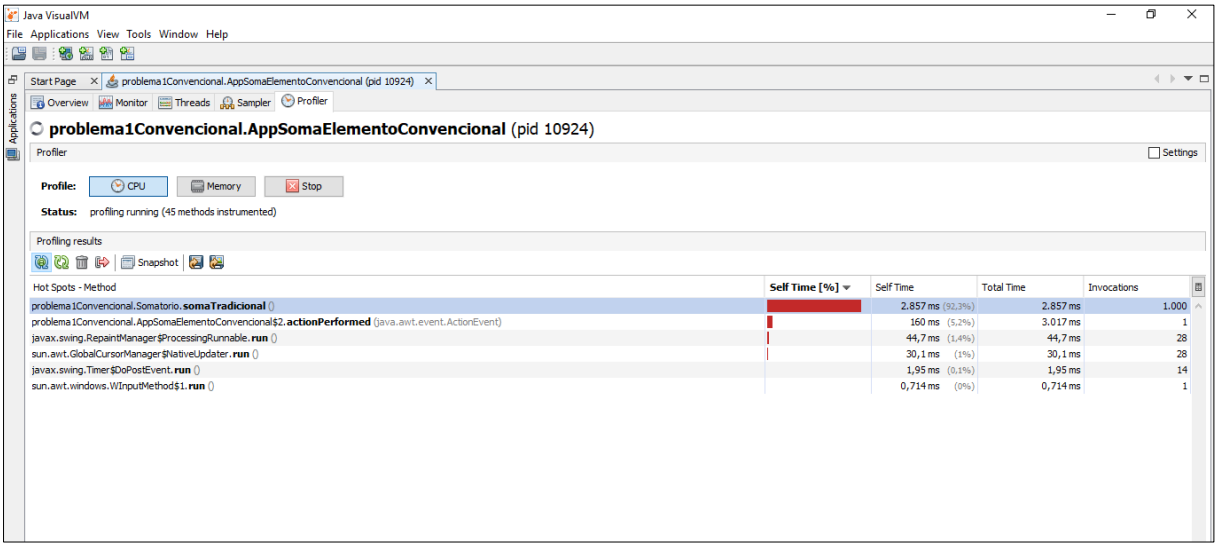
Figura 8 - VisualVm, *Sampler* memória.



Fonte: Adaptada, Oracle (2019)

Na aba *Profiler*[7] (Figura 9), podemos analisar o desempenho da CPU e memória com uma maior precisão. As classes e métodos serão instrumentados, ou seja, toda vez que uma classe ou método for invocado, eles serão registrados, todavia, isso pode afetar a execução da aplicação pois os *bytecodes* de suas classes serão modificados. As *threads* emitem um evento de entrada e saída, ambos contendo informações de data e hora, os dados são processados em tempo real. Um tutorial de como utilizar a ferramenta está disponível em (ORACLE, 2019, on-line).

Figura 9 - VisualVm, aba *Profiler*.



Fonte: Adaptada, Oracle (2019)

[7] VisualVm – Profiler: https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/profiler.html

3 DESENVOLVIMENTO DAS APLICAÇÕES

Buscando responder os questionamentos apresentados anteriormente, foram analisados três problemas conhecidos da computação. Existem algumas maneiras de analisar o funcionamento de software ou aplicação, seja ela de forma parcial ou total. Levando em consideração os problemas abordos utilizamos uma ferramenta que analisou as aplicações em tempo de execução, com foco no desempenho do programa. No decorrer da Seção, iremos descrever como foram implementados os problemas abordados, as ferramentas utilizadas, e as métricas analisadas em cada problema.

3.1 FERRAMENTAS UTILIZADAS

Para a criação das aplicações utilizaremos a ferramenta IDE Eclipse 2018 – 09 (4.9), JRE (*Java Runtime Environment*), JDK (*Java Development Kit*) 1.8.0_221 e o WindowBuilder[8] 1.9.2. Para a realização dos testes, temos a VisualVM (*Profiling*) para realizar análise do sistema em tempo de execução; ela está disponível no próprio JDK e o *notebook Samsung* rv411, sua configuração está presente no Quadro 5 e Sistema operacional Windows 10 *Home*. Trata-se de um computador de uso pessoal, com configurações simples. Por fim os resultados serão apresentados em tabelas.

Quadro 5 - Configuração Notebook

Configuração do Notebook	
Sistema Operacional	Windows 10 Home
Processador	Intel (R) Core (TM) i3 CPU M380 2.53 GHz
Memoria Principal (RAM)	4,00 GB
Tipo do Sistema	Sistema Operacional de 32 bits, processador com base x64

Fonte: Autor

3.2 PROBLEMAS ABORDADOS

Foram escolhidos três problemas para verificar se há ganho em tempo de execução quando utilizamos características funcionais na programação orientada a objetos. Neste aspecto

[8] WindowBuilder – ferramenta de desenvolvimento de interface gráfica. Mais informações podem ser encontradas em: https://www.eclipse.org/windowbuilder.

temos a utilização das expressões lambda, que estão presentes a partir do Java SE 8, em relação à programação convencional. Visando um melhor entendimento da aplicação das expressões lambda no Java, foram implementados problemas já conhecidos, foram eles: Problema 1 – Soma dos elementos em um vetor; Problema 2 – Sequência de Fibonacci recursiva; Problema 3 – Fatorial Iterativo. Como fonte de inspiração recorremos aos artefatos encontrados durante a fase de pesquisa, para o entendimento de cada problema.

- Problema 1 – Soma dos elementos em um vetor: Este procedimento é simples, dado um vetor de inteiros, qual o valor da soma dos elementos contidos nele? Este problema é uma adaptação dos exemplos contidos nos capítulos 7.4 e 17.3, no livro de: Deitel e Deitel (2017).
- Problema 2 – Sequência de Fibonacci recursivo: Sequência de números inteiros, iniciada geralmente por 0 e 1, na qual, cada termo subsequente corresponde à soma dos dois anteriores. A sequência pode ser definida recursivamente pela seguinte fórmula:

$$\begin{cases} F_1 = 1 \\ F_2 = 1 \\ F_n = F_{n-1} + F_{n-2} \end{cases}$$

  O problema apresentado é uma adaptação do capítulo 18 do livro de Deitel e Deitel (2017).
- Problema 3 – Fatorial Iterativo: Seja um número natural n, o fatorial do mesmo é representado por $n!$, que é o produto de todos os inteiros positivos menores ou iguais a n. A inspiração para este problema encontra-se no capítulo 18 do livro de Deitel e Deitel (2017).

3.3 IMPLEMENTAÇÃO DOS PROBLEMAS

Os problemas mencionados serão implementados de duas formas: O modelo 1, se deu em sua forma convencional, utilizando o paradigma orientado a objetos e imperativo. No modelo 2, utilizamos o novo recurso do Java SE 8, as expressões lambda, dando características da PF ao Java. Os problemas foram criados de formas independentes, cada um com sua própria interface, uma para problemas solucionados na forma tradicional, e outra para as expressões lambda. Todas as interfaces dispõem de um campo que irá mostrar o tempo de execução de

cada problema para uma entrada. Neste aspecto, este campo será para simples ilustração, sendo ele não computado na realização dos testes. Os tempos são capturados no ato da execução dos métodos, isso é possível utilizando o método *now()*, da classe *Instant*. Ele obtém o instante atual do relógio do sistema, desta forma temos um intervalo inicial, antes do método ser iniciado, e o intervalo final, após a execução do método. Com a captura destes valores utilizamos o método *between(tempoInicio, tempoFinal)*, que recebe dois argumentos da classe *Duration*. Com isso podemos medir a quantidade de tempo que pode ser expressa em até nanossegundos, a Listagem 8 ilustra essa descrição. Para cada problema foi criado um Java Project. Nele teremos dois pacotes, um contendo uma solução convencional e outro utilizando expressões lambda. Cada pacote tem duas classes sendo uma para a interface gráfica e outra com a implementação do problema proposto. O código das implementações foi depositado no GitHub[9].

Listagem 8 - Medindo o tempo de execução de um algoritmo

```
115
116                 Instant tempoInicial = Instant.now(); // Capturando tempo inicial
117                 for (int j = 0; j < n; j++) {
118
119                     result = fl.fatLambda(i);       // Método chamado
120                 }
121                 Instant tempoFinal = Instant.now(); // Capturando tempo Final
122
123                 String resultadoFibLambda = String.valueOf(result);
124                 txtField_Result.setText(resultadoFibLambda);
125
126                 double tempoDurancao = Duration.between(tempoInicial, tempoFinal).toMillis(); // Tempo pecorrido.
```

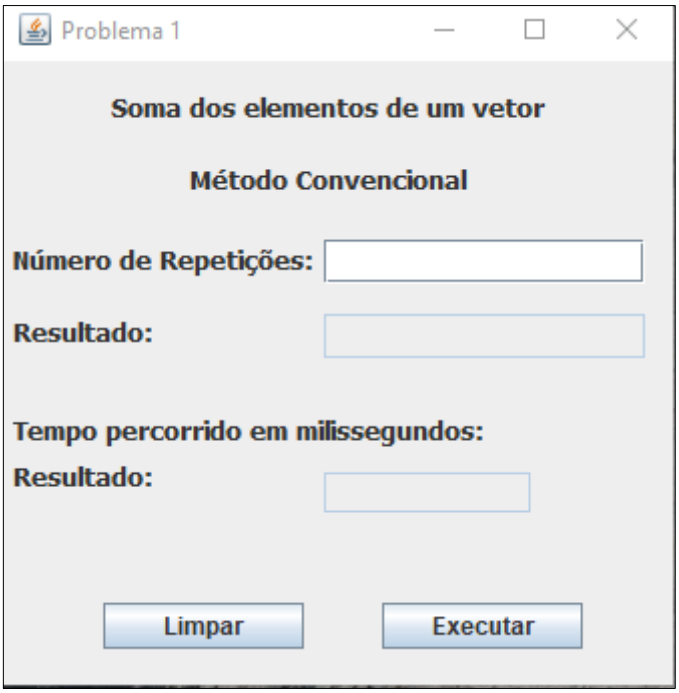
Fonte: Adaptada, Deitel e Deitel (2017)

3.3.1 Problema 1

As implementações do Problema 1 são ilustradas nas Figuras 10 (método convencional) e Figura 11 (método lambda). Elas seguem o mesmo padrão, a diferença está na forma como são calculados os elementos de um vetor.

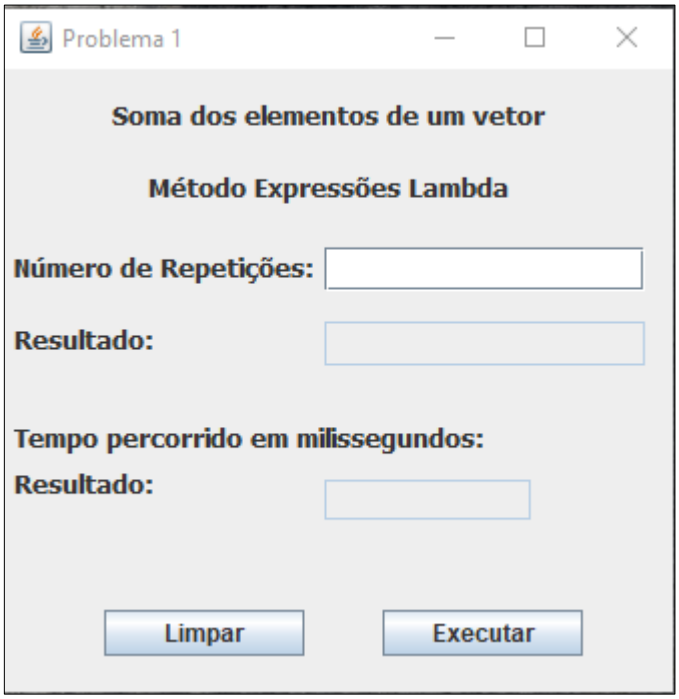
[9] Repositório online, link: https://github.com/bilnp29/tcc

Figura 10 - Problema 1 (método convencional)



Fonte: Autor

Figura 11 - Problema 1 (método lambda)



Fonte: Autor

A funcionalidade de ambas implementações segue o mesmo procedimento, ao pressionar o botão *Executar*. Ambos realizam a seguinte tarefa: Captura o valor de entrada, número de repetição, quantidade de vezes que o método será chamado e faz uma chamada ao

método das classes respectivas. Ao final da execução temos o resultado da soma dos elementos em um vetor e o tempo percorrido em milissegundos. O botão *Limpar* apaga os valores nos campos da implementação. O botão "Executar" do método convencional Figura 10 faz chamada ao seguinte método: *somaTradicional()* que está presente na classe *Somatorio()* realiza a soma dos elementos a partir de um vetor de inteiro. Segue na Listagem 9 sua implementação.

Listagem 9 - Método: *SomaTradicional().*

```
13    public int somaTradicional() {
14        int soma = 0;
15        for (int i = 1; i < v.length; i++) {
16            soma += v[i]; // Soma dos elemantos
17        }
18        return soma;
19    }
```

Fonte: Adaptada, Deitel e Deitel (2017).

O método descrito é autoexplicativo, sua utilização é convencional. A ideia é verificar se a implementação é superior a da Listagem 10, em relação ao tempo de execução.

Listagem 10 - Método: *SomarElementosLambda().*

```
14    public int somarElementosLambda() {
15        return IntStream.of(v).reduce(0, (x,y) -> x + y);
16    }
```

Fonte: Adaptada, Deitel e Deitel (2017)

O método, *SomarElementosLambda()* que está presente na classe *SomatorioLambda(),* é inicializado quando o usuário clica o botão *Executar* do método lambda (Figura 11). O procedimento é realizado em uma linha, ele será executado por meio de uma sequência de passos. Primeiro vamos inicializar o fluxo, utilizando o método *IntStream static of*, recebe o vetor e retorna um *IntStream* para processar os valores do array. A interface *IntStream*, presente no pacote (java.util.stream), é especialista em manipular valores do tipo *int*. Continuando o processo, fazemos uma chamada ao método terminal *reduce (int identity, IntBinaryOperator op).* Este método recebe dois argumentos, o primeiro argumento (0) é o valor para iniciar a redução, e o segundo, um objeto que implementa a interface funcional *IntBinaryOperator* (pacote.java.util.function). A expressão lambda: (x,y) -> x + y, implementa o método

applyAsInt da interface, que recebe dois valores do tipo *int* e executa o cálculo com os valores. O mesmo acontece da seguinte forma: Imagine um vetor de inteiros vt = {2,4}, vamos realizar a soma dos elementos deste vetor.

- Na primeira chamada a *reduce*, o valor do lambda x = 0, assumindo o valor do primeiro argumento de *reduce*, e o valor do lambda y =2 o primeiro elemento do fluxo, sendo acima temos a operação soma que é 2 (0 + 2).
- Na próxima chamada ao método, o valor do lambda x = 2, o resultado do primeiro cálculo, e o lambda y = 4, o próximo elemento do fluxo. Logo, temos que o resultado da soma que é 6 (2 + 4).

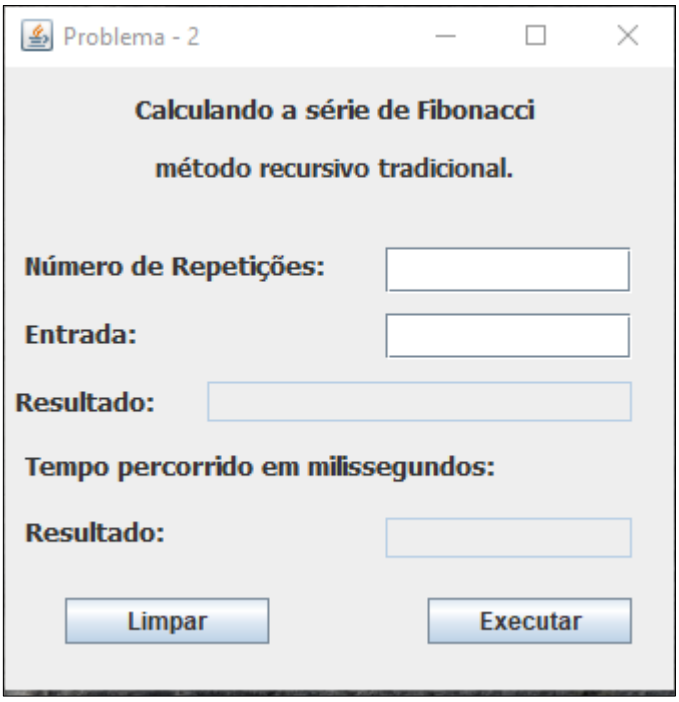
Perceba que o parâmetro lambda x faz papel de um acumulador, já o y recebe os elementos do vetor (DEITEL e DEITEL, 2017).

3.3.2 Problema 2

A sequência de Fibonacci já é conhecida na comunidade e sua escolha é proposital visto que implementada recursivamente, o tempo do cálculo e o número de chamadas aumenta substancialmente, sua implementação encontra-se na Listagem 11. Foi pensando nisso que elaboramos a seguinte implementação. A Figura 12 representa o método recursivo tradicional, já a Figura 13 o método recursivo utilizando expressão lambda. Elas seguem a mesma ideia do Problema 1.

O seu funcionamento é simples, basta inserir um valor nos campos: número de repetições e entrada, o campo número de repetições representa a quantidade de vezes que o método será executado já o campo entrada, o valor de Fibonacci a ser encontrado. Depois é só pressionar o botão *Executar* para as respectivas interfaces. No entanto, o modelo apresentado tem suas limitações. Um valor inserido no capo entrada superior a 90, gera erro no resultado calculado, um estouro de pilha. O botão *Limpar* tem a mesma funcionalidade do Problema 1.

Figura 12 - Problema 2, Fibonacci recursivo.



Fonte: Autor

Quando o botão *Executar* é pressionando Figura 12, ele vai capturar a entrada, converter a String em inteiro e chamar o método *fibonacci(int i)* com o parâmetro capturado. O método em questão é implementado na sua forma tradicional (Listagem 11).

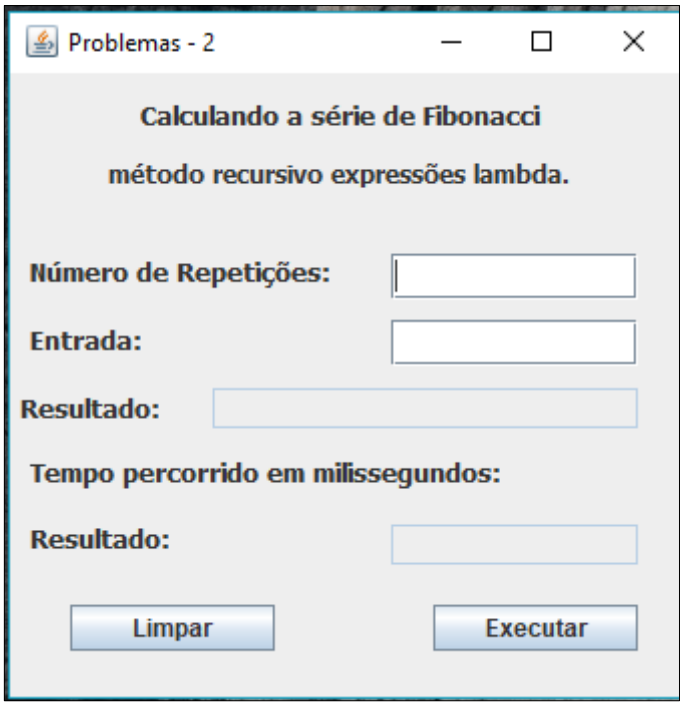
Listagem 11 - Problema 2 (método convencional)

```
5    public long fibonacci(int i) {
6        if (i <= 2) {
7            return 1;
8        }
9        return fibonacci(i - 1) + fibonacci(i - 2);
10   }
```

Fonte: Adaptada, Deitel e Deitel (2017)

Para o problema apresentado existem dois casos básicos no cálculo de Fibonacci: fibonacci(1) e fibonacci(2), nos quais o teste é realizado (linha 6), seja i <=2 retorna 1, Caso i > 2 o passo da recursão é acionado gerando duas chamadas recursivas (linha 9). Neste aspecto elas serão geradas até o caso base ser acionado. Outra maneira de implementar a série é apresentada na Figura 13 e Listagem 12.

Figura 13 - Problema 2, Fibonacci recursivo.



Fonte: Autor

Quando o usuário pressiona o botão *Executar* Figura 13, o mesmo captura o valor de entrada, converte o valor em inteiro e aciona o método fibonacciLambda(int i) repassando o argumento convertido.

Listagem 12 - Problema 2 (método lambda)

```
7⊖    public long fibonacciLambda(int i) {
8         IntPredicate predicado = value -> value <= 2;
9         if (predicado.test(i)) {
10            return 1;
11        }
12        return fibonacciLambda(i - 1) + fibonacciLambda(i - 2);
13    }
```

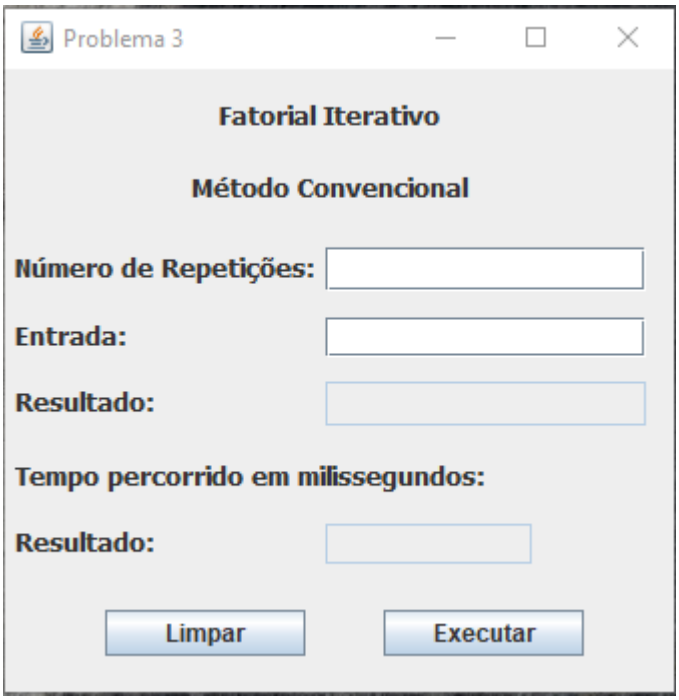
Fonte: Adaptada, Deitel e Deitel (2017)

Temos com isso a inclusão da característica da programação funcional (linha 8) na Listagem 12. O seu funcionamento acontece da seguinte forma: A condição de parada é representada pela variável *predicado* do tipo *IntPredicate*, uma interface funcional, o lambda *(value -> value <=2)* que está presente na variável que implementa o método *test(i)* (linha 9), que recebe um inteiro e retorna um booleano. O valor será avaliado pelo condicional, sendo ele verdade, retorna 1. Caso contrário teremos duas chamadas recursivas na linha (12).

3.3.3 Problema 3

Para o Problema 3 serão utilizadas as seguintes implementações para a realização dos testes. Figura 14, a partir desta interface será calculado fatorial iterativo utilizando uma implementação tradicional. Figura 15, aqui iremos calcular o fatorial iterativo utilizando expressão lambda.

Figura 14 - Problema 3. Fatorial iterativo.



Fonte: Autor

O funcionamento é simples, o usuário insere um valor numérico no campo entrada e no campo número de repetições. O campo número de repetições representa a quantidade de vezes que o método será executado já o campo entrada, o Fatorial a ser encontrado. Este modelo também apresenta suas limitações, sabendo que o resultado do cálculo de um número fatorial cresce muito rápido, logo, sua entrada não pode ser superior a 20. Acreditamos que esta limitação não vai interferir nos testes que serão executados.

Ao pressionar o botão Calcular – C, ele captura os valores de entrada, converte para inteiro e chama o método *fat(i)* da classe *FatorialIterativoConvencional*, sua codificação é ilustrada na Listagem 13.

Listagem 13 - Problema 3 (método convencional)

```
5⊖    public Long fat(int n) {
6         long f = 1;
7         for (int i = 1; i <= n; i++)
8             f *= i;
9         return f;
10    }
```

Fonte: Adaptada, Deitel e Deitel (2017)

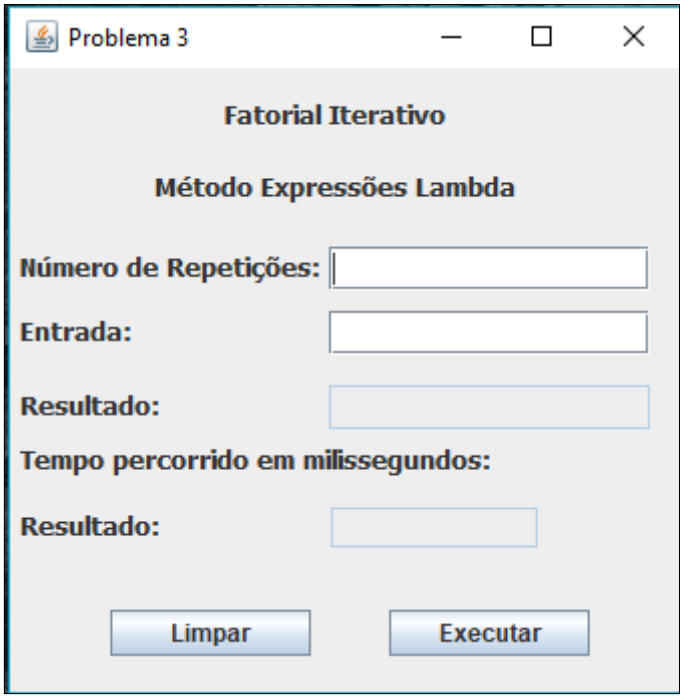
O método apresentado na Listagem 13 representa o cálculo do fatorial de forma convencional. Seu funcionamento ocorre da seguinte forma: na linha 6 temos a variável (f = 1), ela irá armazenar o valor de *n!*; na linha 7, temos uma instrução de repetição *for* que é inicializada em 1 e finalizada em n; no *for*, serão executadas sucessivas multiplicações e armazenada em f; o valor do fatorial de n será retornado ao final da execução (linha 9). Podemos implementá-lo de outra maneira, utilizando as expressões lambda conforme podemos analisar na Listagem 14.

Listagem 14 - Problema 3 (método lambda)

```
6⊖    public long fatLambda(int n) {
7         return LongStream.rangeClosed(1, n).reduce(1, (x, y) -> x * y);
8     }
```

Fonte: Adaptada, Deitel e Deitel (2017)

Figura 15 - Problema 3. Fatorial iterativo.



Fonte: Autor

Quando o usuário pressionar o botão *Executar*, serão capturados os valores de entrada, o programa deve convertê-los para inteiro e chamar o método *fatLambda(i)* da classe *FatorialLambda.* Sua execução é descrita em uma linha, encadeando métodos. Inicialmente temos o *LongStream* uma interface funcional equivalente ao *IntStream,* para o tipo primitivo *long*, que contém o método *rangeClosed (a, b)*, que cria uma sequência ordenada de valores, apresentando dois argumentos que indicam o início e o fim da sequência. A partir da sequência inicializada, o método é encadeado ao *reduce (a, op)*. Sua funcionalidade segue o mesmo padrão do Problema 1, neste caso o método vai realizar a operação de multiplicação.

4 MÉTRICAS AVALIADAS

Para analisar se existem diferenças significativas em relação ao tempo de execução do programa, no qual uma das implementações utilizou expressão lambda, em relação às soluções convencionais utilizando a mesma linguagem, foram adotadas as seguintes métricas para a realização dos testes: tempo de execução de cada método e tempo total da aplicação. Foram selecionadas com base nas literaturas analisadas, no que se referem a testes em sistemas ou codificações específicas. Para testar as métricas acima vamos utilizar a ferramenta VisualVM. Outra métrica analisada é a quantidade de linhas de código em cada método desde sua assinatura até a chave final.

4.1 TEMPO DE EXECUÇÃO

O tempo de execução de um programa ou código, refere-se à velocidade em que o programa é executado, tempo médio para solucionar ou apresentar uma resposta para um problema. Esta métrica irá dimensionar se os problemas aqui testados, apresentam diferenças no tempo de execução. Foram analisados o tempo médio em milissegundos dos métodos de cada solução proposta para o problema, como também o tempo gasto da execução da tarefa como um todo.

4.2 QUANTIDADE DE LINHAS NO CÓDIGO

Ao longo deste trabalho, foram discutidas as questões de legibilidade no código Java, quando utilizadas as expressões lambda e a diminuição da quantidade de linhas de código para realizar uma tarefa. Vários autores destacaram este ganho quando utilizamos este recurso. É comum utilizar esta métrica (quantidade de linhas no código) para avaliar um código gerado. Contudo, com a evolução da engenharia de software, a criação de novos paradigmas e a evolução das linguagens, há quem diga que este parâmetro não é mais relevante (SEABRA, DRUMMOND e GOMES, 2018, p. 63). Contudo, vamos utilizá-lo devido o destaque dado pelos autores levando em consideração que é uma característica importante da biblioteca.

5 TESTES

Após a definição dos problemas e as métricas a serem avaliadas, foi desenvolvido um roteiro para a execução dos testes. Preparação do ambiente: para avaliar melhor os problemas, alguns processos e programas do sistema operacional foram desativados. Só os programas destinados aos testes estiveram abertos. Aplicação e Profiling VisualVM. Para cada problema temos duas soluções, sendo assim foi gerado o .jar de cada uma delas, desta forma temos 6 aplicações ao total. Os testes foram executados de forma sequencial. Problema 1; Problema 2 e Problema 3, iniciando com método convencional depois o método lambda.

No Problema 1 foram realizados seis testes, três no método convencional e três no método lambda, com as seguintes entradas: número de repetições, 1.000, 5.000 e 10.000. Baseado na execução, foi capturada a média do tempo de execução de cada teste, em relação ao tempo percorrido. Para isso, utilizamos a seguinte fórmula:

Equação 1

$$\mathcal{M}t = {}^{\mathcal{T}e}/_{\mathcal{N}r}$$

Fonte: Autor

$$\mathcal{M}t = Média\ do\ tempo\ de\ execução$$
$$\mathcal{T}e = Tempo\ de\ execução$$
$$\mathcal{N}r = Número\ de\ repetição$$

Isso para um vetor de 1.000.000 de posições, pré-estabelecido. A entrada “número de repetições” foi aplicada para ambos os métodos.

Foi realizado o mesmo procedimento para o Problema 2, com as seguintes entradas: Fibonacci de 25, 35 e 45. Executados 100 vezes, primeiro testado o método convencional e depois o método lambda.

O Problema 3, seguiu o mesmo procedimento dos problemas anteriores para as respectivas entradas: número de repetições, 1.000, 5.000 e 10.000 para calcular o fatorial de 20. Utilizamos a Equação 1 para calcular a média de execução do problema.

Explicitadas as entradas que foram aplicadas a cada teste, temos uma demonstração de como foi realizado. O procedimento aplicado é algo comum a todos os testes, seguiu uma sequência de passos. Sendo assim vamos ilustrar um passo a passo, referente ao Problema 1 método convencional. Ao total foram feitos 18 testes, são 3 problemas e para cada um deles

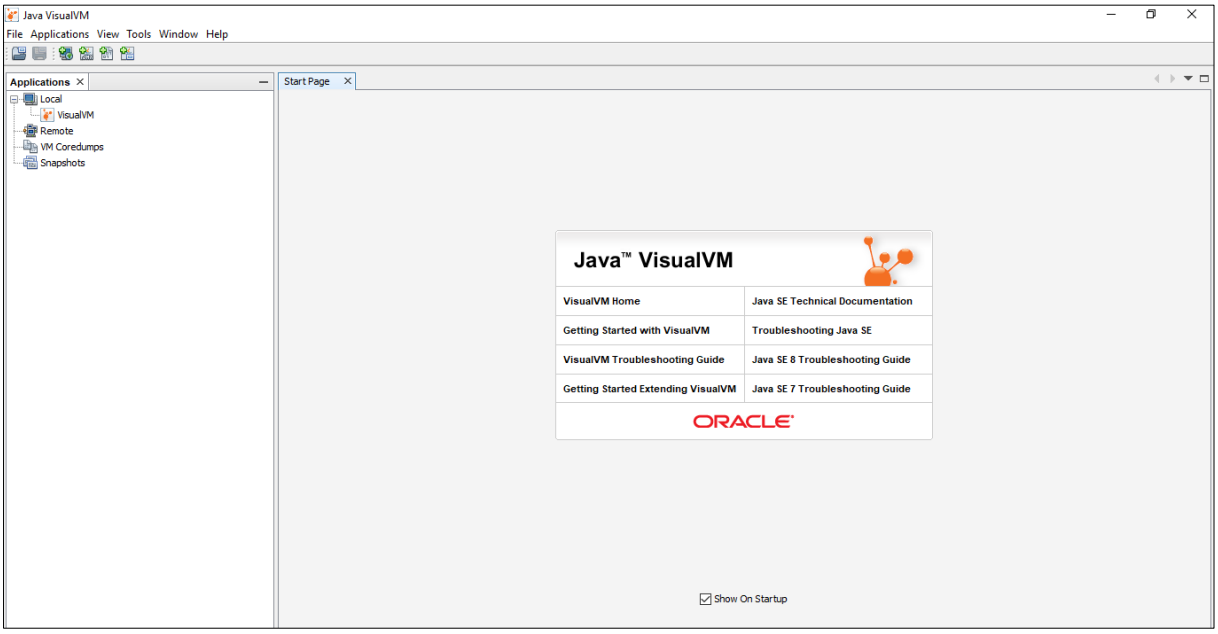
temos 2 aplicações que foram testadas 3 vezes. Desta forma temos uma maior confiabilidade na apresentação dos resultados.

5.1 PASSOS PARA A REALIZAÇÃO DO EXPERIMENTO

1. Conferir estado da máquina: Fechar programas e processos que não são necessários para o funcionamento do sistema. Com o gerenciador de tarefas do Windows podemos verificar quais programas estão ativos e o que está sendo executado em segundo plano. O computador deve estar carregado e conectado a energia, caso ele seja um notebook.

2. Abrir o *Profiling* VisualVm. A tela inicial é ilustrada na Figura 16.

Figura 16 - VisualVM, tela inicial.



Fonte: Adaptada, Oracle (2019)

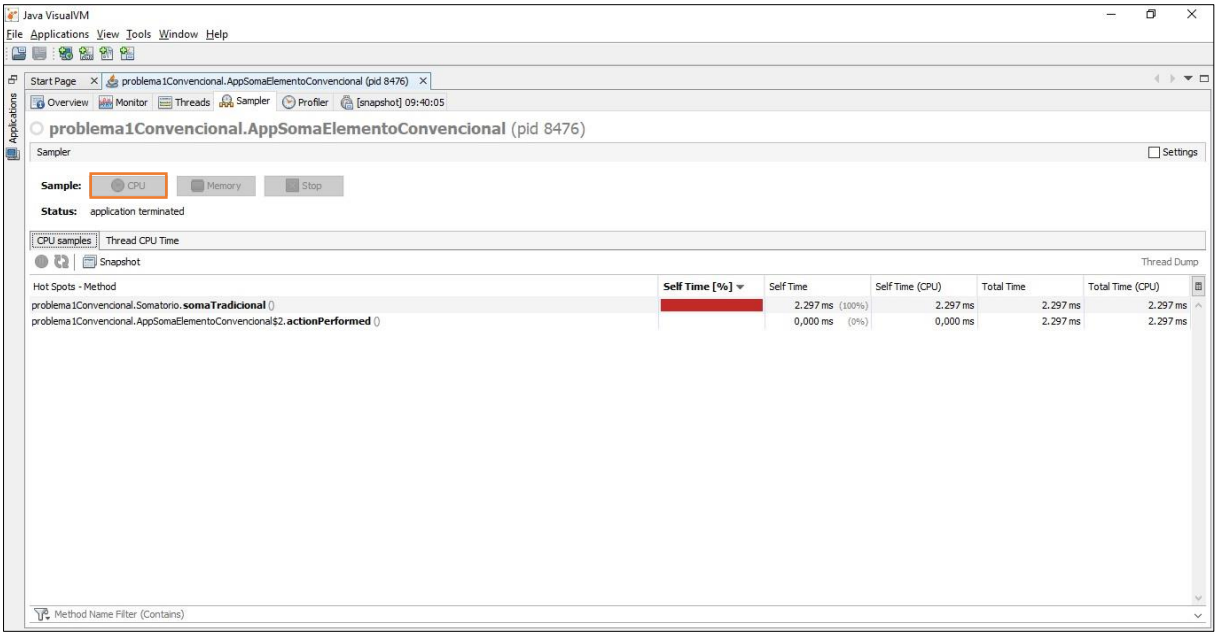
3. Abrir aplicação. Quando aberta, o VisualVM a identifica de imediato.
4. No VisualVm, no painel *Applications*, clique 2 vezes sobre aplicação. Uma aba abrirá, ela se posicionará ao lado da aba *StartPage*. Com ela aberta, por padrão, serão exibidos os dados da aplicação testada (Figura 17).

Figura 17 - VisualVM, aba Overview.

Fonte: Adaptada, Oracle (2019)

5. Com a aba *Sampler* aberta ilustrada na Figura 18, pressione o botão *CPU*, informe as entradas pedidas na aplicação e inicialize a mesma, quando a execução terminar, na aba *CPU Samples*, pressione o botão *Snapshot*, assim vamos capturar dados referentes ao tempo de execução dos métodos testados. Existem casos que o tempo de execução fica abaixo dos milissegundos, sendo assim não é gerado o *Snapshot*.

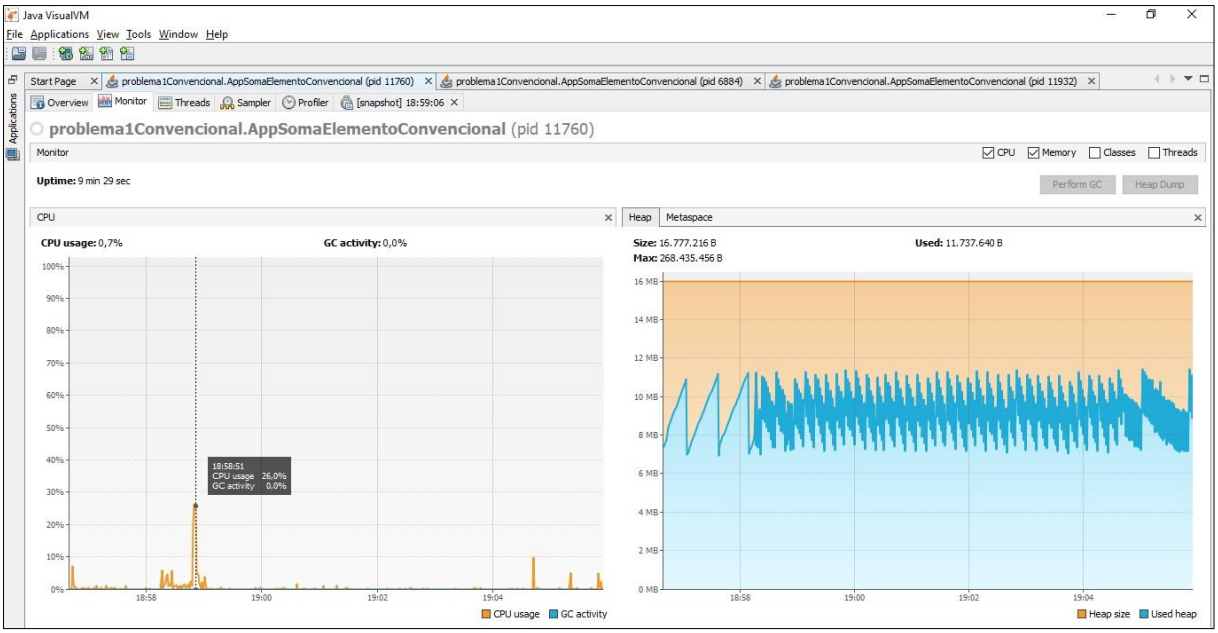
Figura 18 - VisualVM, aba *Sampler*.



Fonte: Adaptada, Oracle (2019)

6. Feche a aplicação e colete as informações presentes no VisualVM.
7. Na aba Monitor ilustrada na Figura 19, podemos capturar os dados referentes à quantidade de *CPU* e memória, alocada e usada.

Figura 19 - VisualVM, aba *Monitor*.



Fonte: Autor

8. Na aba *Sampler*, foram coletados os dados referentes ao tempo de execução que a *CPU* levou para realizar o cálculo. E salvar os dados gerados pelo *Snapshot* ilustrado na Figura 20.

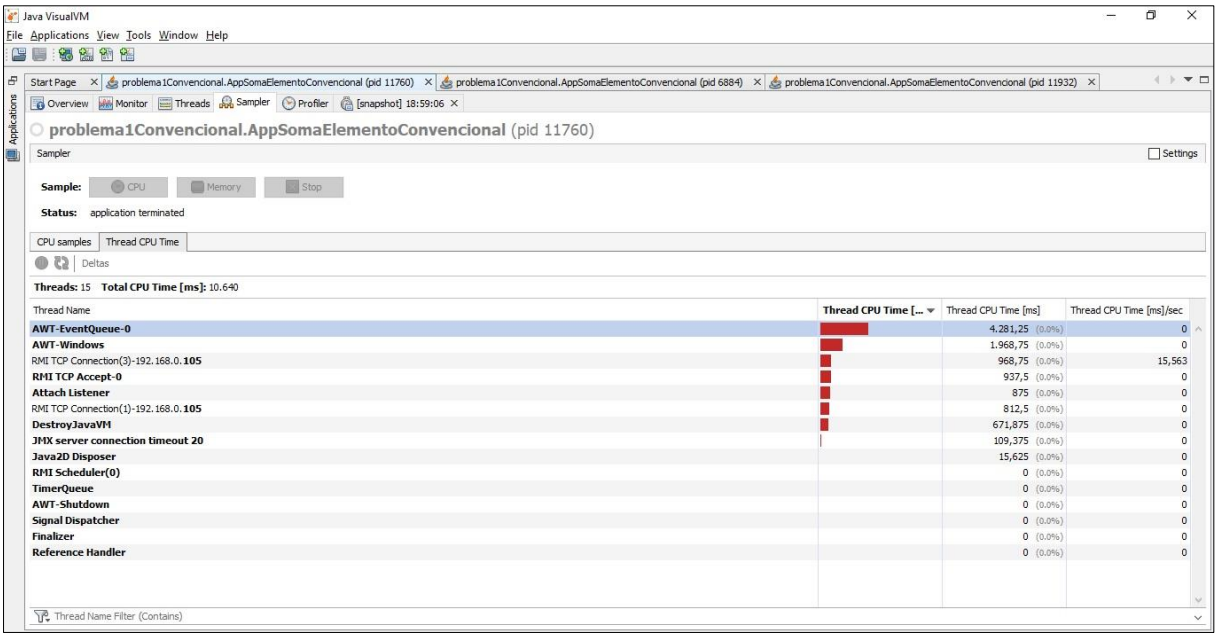
Figura 20 - VisualVM, *Snapshot* demonstração.

Call Tree - Method	Total Time [%]	Total Time	Total Time (CPU)
java.awt.Container.**dispatchEventImpl** ()		2.795 ms (100%)	2.795 ms
java.awt.LightweightDispatcher.**dispatchEvent** ()		2.795 ms (100%)	2.795 ms
java.awt.LightweightDispatcher.**processMouseEvent** ()		2.795 ms (100%)	2.795 ms
java.awt.LightweightDispatcher.**retargetMouseEvent** ()		2.795 ms (100%)	2.795 ms
java.awt.Component.**dispatchEvent** ()		2.795 ms (100%)	2.795 ms
java.awt.Container.**dispatchEventImpl** ()		2.795 ms (100%)	2.795 ms
java.awt.Component.**dispatchEventImpl** ()		2.795 ms (100%)	2.795 ms
java.awt.Container.**processEvent** ()		2.795 ms (100%)	2.795 ms
java.awt.Component.**processEvent** ()		2.795 ms (100%)	2.795 ms
javax.swing.JComponent.**processMouseEvent** ()		2.795 ms (100%)	2.795 ms
java.awt.Component.**processMouseEvent** ()		2.795 ms (100%)	2.795 ms
javax.swing.plaf.basic.BasicButtonListener.**mouseReleased** ()		2.795 ms (100%)	2.795 ms
javax.swing.DefaultButtonModel.**setPressed** ()		2.795 ms (100%)	2.795 ms
javax.swing.DefaultButtonModel.**fireActionPerformed** ()		2.795 ms (100%)	2.795 ms
javax.swing.AbstractButton$Handler.**actionPerformed** ()		2.795 ms (100%)	2.795 ms
javax.swing.AbstractButton.**fireActionPerformed** ()		2.795 ms (100%)	2.795 ms
problema1Convencional.AppSomaElementoConvencional$2.**actionPerformed** ()		2.795 ms (100%)	2.795 ms
problema1Convencional.Somatorio.**somaTradicional** ()		2.795 ms (100%)	2.795 ms
Self time		0,000 ms (0%)	0,000 ms
Self time		0,000 ms (0%)	0,000 ms
Self time		0,000 ms (0%)	0,000 ms
Self time		0,000 ms (0%)	0,000 ms

Fonte: Adaptada, Oracle (2019)

9. Ainda em *Sampler*, na aba *Thread CPU Time*, será capturado o tempo de execução do thread *Awt-EventQueue-o*, referente ao tempo total de execução da aplicação ilustrado na Figura 21.

Figura 21 - VisualVM, imagem *Thread CPU time*.



Fonte: Adaptada, Oracle (2019)

10. Após realizar toda a sequência descrita acima, certifique-se de que tudo foi salvo.
11. Repetimos todos os passos a partir do item 3, respeitando as entradas mencionadas e a sequência proposta.

Ao final todos os dados foram arquivados e computados, o procedimento ocorreu de maneira satisfatória.

6 RESULTADOS

6.1 PROBLEMA 1

Após realizarmos os testes propostos os resultados foram colhidos e analisados. A Tabela 1 agrupa os resultados dos tempos de execução dos métodos propostos e dos testes realizados. Lembramos que cada teste representa uma entrada diferente e que os métodos aqui analisados são: *somaTradicional()* método convencional e *SomarElementosLambda()* método lambda. Podemos observar que tanto no método convencional, quanto no método lambda, os resultados não apresentam grandes diferenças em relação ao tempo de execução de cada um deles.

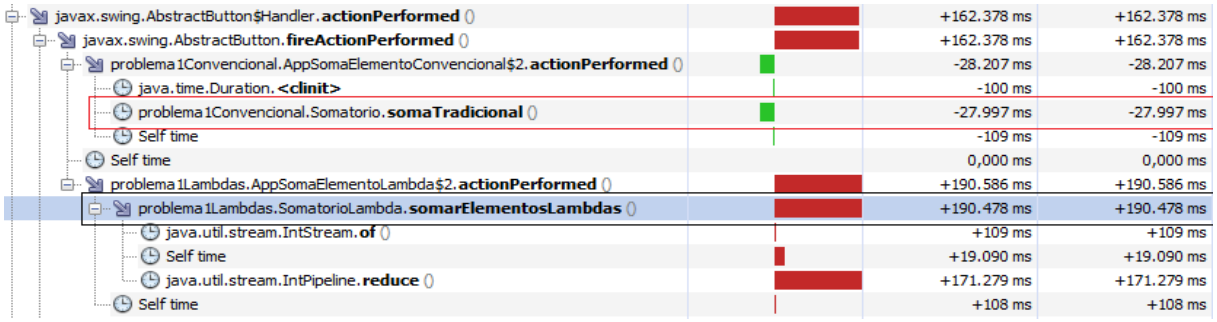
Tabela 1- Resultados obtidos do método (Problema 1).

Problema 1 - Soma dos elementos de um vetor			
Tempo de execução dos métodos propostos.			
	Teste 1	Teste 2	Teste 3
Método Convencional	2,30	2,84	2,80
Método Lambda	18,89	18,86	19,01

Fonte: Autor

A partir dos valores obtidos para a métrica e tempo de execução dos métodos, temos que o método convencional executa em menos tempo. Vale lembrar que o resultado obtido é em milissegundos. Vamos agora comparar os *Snapshot* capturados nos testes, como os resultados não apresentam grandes diferenças vamos utilizar o Teste 3. Com o VisualVM podemos comparar os *Snapshot* salvos. Sendo assim utilizamos este recurso, conforme a ilustração da Figura 22 demonstra comparativamente.

Figura 22 - Comparação dos *Snapshot* (Problema 1, Teste 3).



Fonte: Adaptada, VisualVm (2019)

Os valores acima são os tempos originais. Vale lembrar que os tempos apresentados na Tabela 1 é a média obtida a partir da Equação 1 proposta, descrita anteriormente. A marcação em vermelho faz referência ao método *somaTradicional()*, já a preta, ao método *somarElementosLambda()*. Perceba que a diferença em relação ao tempo de execução entre os dois métodos é de 162.479 milissegundos. Foram contabilizados outros procedimentos que não iremos levar em consideração. Sendo assim, utilizando a equação 1 temos que a diferença é de 16,25 milissegundos.

Para o tempo de execução total da aplicação temos os seguintes resultados apresentados na Tabela 2

Tabela 2 - Resultados obtidos da aplicação do Problema 1

Problema 1 - Soma dos elementos de um vetor			
Tempo de execução total das aplicações			
	Teste 1	Teste 2	Teste 3
Aplicação Convencional	2,88	3,03	2,86
Aplicação Lambda	19,36	19,06	19,11

Fonte: Autor

Podemos perceber que não houve grandes alterações em relação ao tempo de execução do aplicativo para os métodos inseridos em cada aplicação. Neste aspecto, podemos deduzir que a aplicação não interfere nos resultados obtidos na Tabela 1.

Em relação a quantidade de linhas para resolver o problema, temos que o método lambda realiza o procedimento com uma quantidade menor, de acordo com a Tabela 3, validando os argumentos dos autores mencionados durante o trabalho.

Tabela 3 - Quantidade de linhas (Problema 1)

Problema 1 - Soma dos elementos de um vetor	
Quantidade de linhas utilizada.	
	Linha
Método Convencional	5
Método Lambda	1

Fonte: Autor

6.2 PROBLEMA 2

A Tabela 4 representa o tempo de execução dos métodos *fibonacci(int i)* e *fibonacciLambda(int i),* método convencional e método lambda como resultados para o Problema 2. Após a realização dos testes, os resultados foram agrupados e comparados. Neste sentido podemos destacar que o método convencional foi executado em menos tempo. Lembrando que os tempos aqui apresentados são em milissegundos e foi utilizada a equação 1, destacada anteriormente.

Tabela 4 – Resultado obtido do método (Problema 2)

Problema 2 - Fibonacci Recursivo			
Tempo de execução, dos métodos propostos.			
	Teste 1	Teste 2	Teste 3
Método Convencional	1,15	47,06	5667,01
Método Lambda	1,02	64,12	7745

Fonte: Autor

Para o tempo de execução total da aplicação, que é descrito na Tabela 5, temos que ele não demostrou maiores diferenças em relação aos métodos testados anteriormente. Desta forma, a aplicação não influencia nos resultados obtidos, conforme observa-se na Tabela 4.

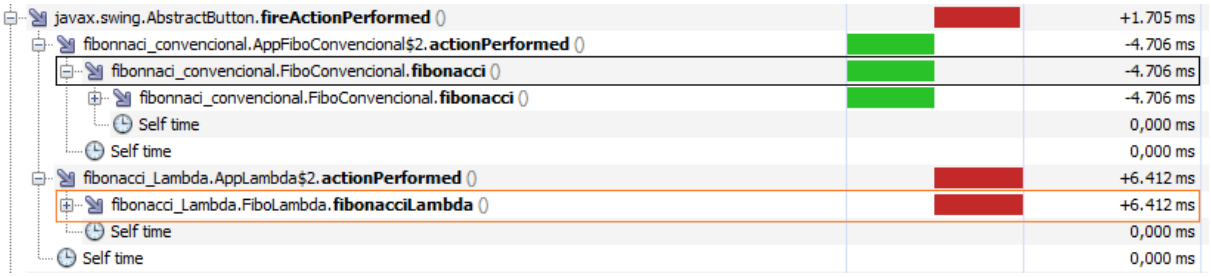
Tabela 5 - Resultados obtidos, aplicação (Problema 2)

Problema 2 - Fibonacci Recursivo			
Tempo de execução total das aplicações			
	Teste 1	Teste 2	Teste 3
Método Convencional	5	49,84	5669,84
Método Lambda	4,6875	67,34	7747,03

Fonte: Autor

Utilizando o VisualVM, vamos comparar os resultados obtidos no teste 2, sendo ele escolhido aleatoriamente. Vale lembrar que foram utilizadas entradas diferentes em cada teste, eles foram repetidos 100 vezes, legitimando os resultados obtidos.

Figura 23 - Comparação dos Snapshot (Problema 2, Teste 2)



Fonte: Autor

A Figura 23 ilustra o tempo de execução de cada método em sua forma original. A diferença do tempo de execução entre os dois métodos é de 17,06 milissegundos, isso equivalente a 0,017 segundos, ou seja, mais uma vez o método convencional é executado mais rápido do que o método lambda. Em relação a quantidade de linhas utilizadas para solucionar o problema temos que o método convencional apresentou menos linhas para solucionar o problema.

Tabela 6 - Quantidade de linhas (Problema 2)

Problema 2 – Fibonacci Recursivo	
Quantidade de linhas utilizada.	
	Linha
Método Convencional	5
Método Lambda	6

Fonte: Autor

6.3 PROBLEMA 3

Os resultados obtidos para o tempo de execução dos métodos, são descritos na Tabela 7. Neste aspecto foram testados os métodos *fat(i)* e *fatLambda(i)*, método convencional e método lambda, respectivamente.

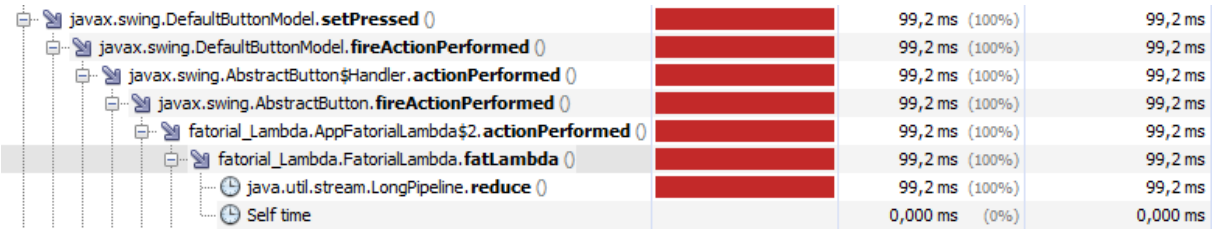
Tabela 7 - Resultados obtidos, método (Problema 3)

Problema 3 - Fatorial Iterativo			
Tempo de execução, dos métodos propostos			
	Teste 1	Teste 2	Teste 3
Método Convencional	0,0000	0,0000	0,0000
Método Lambda	0,0000	0,0000	0,0099

Fonte: Autor

De acordo com o resultado, não podemos definir qual método executou em menos tempo. Lembrando que o tempo é medido em milissegundos. Sendo assim definimos que para a métrica proposta, os métodos são equivalentes. Em relação ao *snapshot*, ele só foi gerado para o Teste 3 métodos lambda, os outros testes ficaram próximos de zero. A Figura 24 ilustra o *snapshot*.

Figura 24 – Snapshot, método lambda Teste 3, Problema 3



Fonte: Autor

O tempo, apresentado na imagem, está em sua forma original. Ele está associado ao método *reduce*, já explicado aqui neste trabalho. Em relação ao tempo total da execução da aplicação, a Tabela 8 representa estes valores de forma agrupada, facilitando o entendimento.

Tabela 8 - Resultados obtidos, aplicação (Problema 3)

Problema 3 - Fatorial Iterativo			
Tempo de execução total das aplicações			
	Teste 1	Teste 2	Teste 3
Método Convencional	0,3750	0,0624	0,0359
Método Lambda	0,3750	0,0813	0,0328

Fonte: Autor

Como podemos perceber, o mesmo continua abaixo de 1 milissegundo. Desta forma, elas não interferem no tempo de execução dos métodos. Em relação a quantidade de linhas

necessárias para resolver o problema, o método lambda leva vantagem, com apenas uma linha o problema é resolvido, visto na Tabela 9.

Tabela 9 - Quantidade de linhas (Problema 3)

Problema 3 – Fatorial Iterativo	
Quantidade de linhas utilizada.	
	Linha
Método Convencional	5
Método Lambda	1

Fonte: Autor

6.4 ANÁLISE DOS RESULTADOS

Após a realização dos testes e apresentação dos resultados, podemos concluir que para alguns problemas, o uso das expressões lambda mostrou um tempo de execução significativamente maior em relação as soluções convencionais.

Os problemas aqui propostos são conhecidos computacionalmente e existem formas diferentes de implementar cada um deles. Sua escolha foi proposital devido a quantidade de cálculos que são necessários para resolver o problema e o tempo que isso leva. Utilizar o método lambda é uma forma de deixar o código mais enxuto. No entanto, isso não aconteceu no Problema 2. Acreditamos que a solução apresentada, em termos de codificação, pode ser melhorada, isso porque temos uma verificação a mais, linha 9 método *fibonacciLambda(i)*, para resolver o problema.

A Oracle, em um documento online[10], sugere em que momento podemos utilizar as expressões lambda. Cabe ao programador saber quando adotar cada método, neste sentido utilizar o método lambda, aqui apresentado, requer conhecimento da API e mínimos conhecimentos do paradigma funcional. Um exemplo disso é encontrado na solução do Problema 3 (método lambda). Para resolver tal problema é necessário saber como cada método realiza um procedimento. Quando temos 2 métodos para realizar a tarefa, ela é dividida em duas partes, a primeira, método *rangeClosed (a, b),* cria uma sequência de valores ordenados e a segunda, método *reduce (a, op),* reduz a sequência em um valor.

[10] Expressões Lambda - https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html.

Sendo assim, portanto, é necessário que o programador compreenda os conceitos da programação funcional do Java 8 e entenda o funcionamento da API, tendo em vista uma programação declarativa, utilizando encadeamento de funções para solucionar um problema.

7 CONCLUSÃO

A Linguagem Java SE 8 introduziu as expressões lambda. Encontrou-se na literatura que a adição deste recurso trouxe alguns ganhos, tais como: nível de abstração mais alto, robustez (multiparadigma) e facilidade na utilização do processamento em paralelo. Entretanto, como já mencionado anteriormente, sua utilização requer do programado conhecimento de como e quando utilizá-las, de forma a obter os ganhos mencionados acima.

Entretanto, verificou-se através dos testes realizados, que o uso das expressões lambda na Linguagem Java SE 8, em alguns casos, podem aumentar o tempo de execução.

Como trabalhos futuros sugerimos um estudo da utilização das expressões lambda no processamento em paralelo, assim como verificar se existem diferenças na utilização das expressões lambda do Java em relação a outras linguagens que as suportem.

REFERÊNCIAS

ALLEN, Christopher; MORONUKI, Julie. **Haskell programmig from frist principles**. 1. ed. [*S. l.*]: Allen and Moronuki Publishing, 2016. 1196 p. v. 1.

AMORIN, GABRIEL NOVAIS. **Java Performance**: Aprimorando o desempenho de aplicações. [*S. l.*], 2014. Disponível em: https://www.devmedia.com.br/java-performance-aprimorando-o-desempenho-de-aplicacoes/ 31277. Acesso em: 16 dez. 2019.

ÁVILA, Augusto Vieira. **PROGRAMAÇÃO FUNCIONAL E REATIVA APLICADA AO DESENVOLVIMENTO DE INTERFACES COM O USUÁRIO EM APLICAÇÕES PARA WEB**. Orientador: Prof. Leandro José Komosinski. 2017. Trabalho de conclusão de curso (Bacharel em Sistemas de Informação) - UNIVERSIDADE FEDERAL DE SANTA CATARINA, Florianópolis, 2017.

BIONDO, GIOVANI. **UM PROCESSO DE CONVERSÃO DE SISTEMAS LEGADOS PROCEDURAIS PARA ORIENTADO A OBJETOS, DIRECIONADO PELA ARQUITETURA MVC**. Orientador: Prof. Me. Joacir Giaretta. 2017. Trabalho de conclusão de curso (Bacharel em Sistemas de Informação) - UNIVERSIDADE DE CAXIAS DO SUL, BENTO GONÇALVES, 2017.

CARVALHO, Marlon Silva. Como aproveitar ao máximo as vantagens das expressões lambda em seu próximo projeto. **Expressões lambda**: Um novo recurso do java 8, Rio de Janeiro, v. I, ed. 136, p. 18-27, 2015.

DEITEL, Paul; DEITEL, Harvey. **Java**: como programar. 8. ed. São Paulo: Pearson Prentice Hall, 2010.

DEITEL, Paul; DEITEL, Harvey. **Java**: como programar. 10. ed. São Paulo: Pearson Education do Brasil, 2017.

DIVERIO, Tiarajú Asmuz; MENEZES, Paulo Blauth. **Teoria da computação**: máquinas universais e computabilidade. 1. ed. Porto Alegre: Sagra Luzzato, 1999.

GASPAROTTO, HENRIQUE MACHADO. **Os 4 pilares da Programação Orientada a Objetos**. [*S. l.*], 2014. Disponível em: https://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264. Acesso em: 16 dez. 2019.

MEDEIROS, Higor. **Profiling**: Como analisar Aplicações Java. [*S. l.*], 2015. Disponível em: https://www.devmedia.com.br/profiling-como-analisar-aplicacoes-java/32500. Acesso em: 16 dez. 2019.

NOGUEIRA, André da Silva. **Profiling de aplicações Web**: Estudo comparativo entre aplicações Java Web e aplicações RoR. Orientador: Professor F. Mário Martins. 2014. Dissertação (Mestrado em Engenharia Informática) - Universidade do Minho, Guimarães, 2014.

OLIVEIRA, Alexandre Ponce. **Teste estrutural para aplicações concorrentes em Erlang**. 2017. Tese (Doutor em Ciências - Ciências de Computação e Matemática) - Universidade de São Paulo, São Carlos, 2017.

ORACLE (Califórnia). **JSR 335**: Lambda Expressions for the JavaTM Programming Language. [*S. l.*], 23 out. 2012. Disponível em: https://www.jcp.org/en/jsr/detail?id=335. Acesso em: 16 dez. 2019.

ORACLE (Califórnia). **Projeto Lambda**. [*S. l.*], 2014. Disponível em: http://openjdk.java.net/projects/lambda/. Acesso em: 16 dez. 2019.

ORACLE (Califórnia). **Java Programming Language Enhancements**. [*S. l.*], 2019. Disponível em: https://docs.oracle.com/javase/8/docs/technotes/guides/language/enhancements.html#javase8. Acesso em: 16 dez. 2019.

ORACLE (Califórnia). **Java VisualVM**. [*S. l.*], 2019. Disponível em: https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/. Acesso em: 16 dez. 2019.

ORACLE (Califórnia). **The Java™ Tutorials**: Method References. [*S. l.*], 2019. Disponível em: https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html. Acesso em: 16 dez. 2019.

PAÍSES BAIXOS (Eindhoven). TIOBE. **August 2019 TIOBE Index**: Silly season in the programming language world. [S. l.], Ago 2018. Disponível em: https://www.tiobe.com/tiobe-index/. Acesso em: 29 ago. 2019.

PINHO, Eduardo Gurgel. **Uma Linguagem de Programação Paralela Orientada a Objetos Para Arquiteturas de Memórias Distribuídas**. 2012. Dissertação (Mestre em Ciência da Computação) - Universidade Federal do Ceará, Fortaleza, 2012.

ROSA, Clayton Wilhelm. **Variabilidade de Software em Linguagens Funcionais**: Um Estudo Exploratório em Haskell. Orientador: Prof. Ivonei Freitas da Silva. 2016. Trabalho de conclusão de curso (Bacharel em Ciência da Computação) - Universidade Estadual do Oeste do Paraná, CASCAVEL, 2016.

SEABRA, Rodrigo Duarte; DRUMMOND, Isabela Neves; GOMES, Fernando Coelho. Análise Comparativa de Linguagens de Programação a partir de Problemas Clássicos da Computação. **Revista de Sistemas e Computação**, Salvador, v. 8, n. 1, p. 56-76, Jan/Jun 2018.

SEBESTA, Robert W. **Conceitos de Linguagem de Programação**. 9. ed. Porto Alegre: Bookman, 2011. Disponível em: https://pt.slideshare.net/andersonguitarr/conceitos-de-linguagem-de-programao-9a-edio-robert-w-sebesta-65178010. Acesso em: 16 maio 2019.

SILVA, Luiz Artur Botelho. "**FLIMSY**: UM MIDDLEWARE FUNCIONAL EM SCALA" Orientador: Prof. Nelson Souto Rosa. 2015. Dissertação (Mestre Profissional em Ciência da Computação) - Universidade Federal de Pernambuco, RECIFE, 2015.

SILVEIRA, Paulo; TURINI, Rodrigo. **Java 8 Prático**: lambda, streams e os novos recursos da linguagem. 1. ed. São Paulo: Casa do Código, 2014. Disponível em: http://www.aeaab.com.br/assets/docs/Java_8_Pratico_Lambda_Streams_e_os_Novos_Recursos_da_Linguagem_-_Casa_do_Codigo.pdf. Acesso em: 9 set. 2019.

TAVARES, Aline Laís Gomes; CALDAS, Filipe Cardoso. **Caracterizando a Adoção de Expressões Lambda em Código Java Legado**. Orientador: Prof. Dr. Rodrigo Bonifácio de Almeida. 2017. Trabalho de conclusão de curso (Bacharelado em Ciência da Computação) - Universidade de Brasília, Brasília, 2017.

TEIXEIRA JUNIOR, Jânio Elias. **Um Catálogo De Refatorações Envolvendo Expressões Lambda Em Java**. 2014. Dissertação (Mestre em Ciência da Computação) - Universidade Federal De Santa Maria, Santa Maria, 2014.

TUCKER, Allen B.; NOONAN, Robert E. **Linguagens de programação**: princípios e paradigmas. 2. ed. Porto Alegre: AMGH, 2010.