UNIVERSIDADE FEDERAL DE PERNAMBUCO – UFPE

CENTRO DE INFORMÁTICA – CIN

CURSO DE BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO

KENNEDY EDMILSON CUNHA MELO

REFATORAÇÃO E ADOÇÃO DAS FUNCIONALIDADES DO JAVA 8+: IMPACTOS E BENEFÍCIOS EM PROJETOS OPEN SOURCE E EMPRESARIAIS

RECIFE

2025

KENNEDY EDMILSON CUNHA MELO

REFATORAÇÃO E ADOÇÃO DAS FUNCIONALIDADES DO JAVA 8+: IMPACTOS E BENEFÍCIOS EM PROJETOS OPEN SOURCE E EMPRESARIAIS

Trabalho de Graduação apresentado ao Curso de Engenharia da Computação do Centro de Informática Universidade Federal de Pernambuco como requisito para obtenção do título de bacharel em Engenharia da Computação.

Orientador: André Luís de Medeiros Santos

Áreas: Linguagem de Programação

RECIFE

2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Melo, Kennedy Edmilson Cunha.
Refatoração e adoção das funcionalidades do Java 8+: impactos e benefícios em projetos open source e empresariais / Kennedy Edmilson Cunha Melo. - Recife, 2025.
46 p : il., tab.

Orientador(a): André Luís de Medeiros Santos
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2025.
Inclui referências, apêndices.

1. Java 8. 2. refatoração de código. 3. lambda. 4. streams. 5. Optional. 6. modernização de software. I. Santos, André Luís de Medeiros. (Orientação). II. Título.

000 CDD (22.ed.)

KENNEDY EDMILSON CUNHA EMLO

Refatoração e adoção das funcionalidades do Java 8+: impactos e benefícios em projetos open source e empresariais

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em 05/08/2025 da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Engenharia da Computação.

Aprovado em: 12/08/2025

BANCA EXAMINADORA

_____

Prof. Dr. André Luís de Medeiros Santos (Orientador)

Universidade Federal de Pernambuco

_____

Prof. Dr. Sérgio Castelo Branco Soares (Examinador Interno)

Universidade Federal de Pernambuco

AGRADECIMENTOS

Agradeço aos meus pais que sempre me incentivaram ao estudo e são a razão de eu poder cursar Engenharia da Computação nesta renomada Instituição, em especial ao meu pai Edmilson Cunha de Melo, o qual sempre me fez priorizar o estudo, proporcionando todos os meios necessários a fim da minha conclusão de curso.

Além disso, agradeço ao meu professor orientador André Luís de Medeiros Santos, que sempre me apoiou, teve paciência e compreensão durante todo o processo de elaboração deste trabalho. Também agradeço ao professor Sérgio Soares que fez a revisão final do documento.

RESUMO

A introdução de novos recursos na linguagem Java, a partir da versão 8, representou uma ruptura paradigmática em sua evolução, incorporando elementos da programação funcional ao seu tradicional modelo orientado a objetos. Este trabalho de conclusão de curso propõe-se a investigar os impactos da refatoração de código legado mediante a adoção de funcionalidades como expressões lambda, interfaces funcionais, Streams API, Optional e Method References. A pesquisa, de natureza qualitativa e exploratória, abrange desde a fundamentação teórica até a análise de código real, contemplando tanto projetos open source quanto soluções empresariais. Os resultados evidenciam benefícios expressivos em termos de legibilidade, modularidade e desempenho do software, ao mesmo tempo em que apontam desafios inerentes ao processo de adoção, como a curva de aprendizado - obtida via formulário a desenvolvedores Java - e a compatibilidade com arquiteturas legadas. O estudo reforça a relevância de práticas de engenharia de software conscientes e embasadas na modernização tecnológica, com vistas à sustentabilidade e à qualidade do código.

Palavras-Chave: Java 8; refatoração de código; lambda; streams; Optional; modernização de software.

ABSTRACT

The introduction of new features in the Java language, starting with version 8, represented a paradigmatic break in its evolution, incorporating elements of functional programming into its traditional object-oriented model. This end-of-course work aims to investigate the impacts of refactoring legacy code by adopting features such as lambda expressions, functional interfaces, Streams API, Optional and Method References. The research, of a qualitative and exploratory nature, ranges from theoretical grounding to the analysis of real code, covering both open source projects and enterprise solutions. The results show significant benefits in terms of readability, modularity and software performance, while at the same time pointing out challenges inherent in the adoption process, such as the learning curve - obtained via a form for Java developers - and compatibility with legacy architectures. The study reinforces the importance of conscientious software engineering practices based on technological modernization, with a view to sustainability and code quality.

Keywords: Java 8; code refactoring; lambda; streams; Optional; software modernization.

SUMÁRIO

1. INTRODUÇÃO..... 1

1.1 Evolução da Linguagem Java a partir da Versão 8.....1

1.2 Objetivos..... 2

1.3 Estrutura do Trabalho.....3

2. FUNDAMENTOS.....5

2.1 Interfaces Funcionais.....5

2.2 Lambda Functions..... 7

2.3 Streams API.....9

2.4 Optional..... 11

2.5 Method References.....13

3. REFATORAÇÃO..... 18

3.1 Refatoração de Código Legado com Java 8+.....18

3.2 Motivações e Benefícios..... 18

3.3 Desafios e Limitações..... 19

4. ANÁLISE EM PROJETOS REAIS: ADOÇÃO DAS FUNCIONALIDADES DO JAVA 8+..... 21

4.1 Aplicação das Funcionalidades em Projetos Open Source..... 21

4.2 Avaliação das Funcionalidades em Projetos Open Source.....25

4.3 Aplicação das Funcionalidades em Projetos Empresariais..... 26

4.4 Avaliação das Funcionalidades em Projetos Empresariais.....31

5. ADOÇÃO, EXPERIÊNCIA E CURVA DE APRENDIZADO..... 33

5.1 Perfil dos Participantes e Cenário de Adoção..... 33

5.2 Experiência com as Funcionalidades do Java 8+: Uso, Aprendizado e Desafios..... 34

5.3 Impactos da Adoção: Manutenção, Produtividade e Refatoração.....36

6. CONCLUSÃO..... 39

7. TRABALHOS FUTUROS E LIMITAÇÕES..... 41

REFERÊNCIAS..... 42

APÊNDICE - QUESTIONÁRIO..... 44

1. INTRODUÇÃO

1.1 Evolução da Linguagem Java a partir da Versão 8

Desde seu surgimento em 1995, a linguagem Java consolidou-se como uma das principais tecnologias utilizadas no desenvolvimento de aplicações corporativas, desenvolvimento web, e posteriormente a serviços de nuvem e aplicações móveis. Um dos principais motivos que explicam sua longevidade é a constante evolução da linguagem, que busca acompanhar as transformações do mercado e as novas práticas da engenharia de software [1].

Até a versão 7, o Java possuía uma estrutura marcadamente imperativa e orientada a objetos. O código produzido exigia o uso extensivo de laços de repetição, condicionais explícitas e classes anônimas — o que resultava, muitas vezes, em implementações prolixas e repetitivas. Isso tornava a manutenção de sistemas mais trabalhosa, especialmente em projetos de grande porte, nos quais a clareza e a modularidade do código são fundamentais [2].

A introdução do Java 8, em março de 2014, representou uma inflexão significativa na evolução da linguagem. Essa versão incorporou elementos do paradigma funcional, seguindo uma tendência já presente em linguagens como Scala e Kotlin, que demonstravam vantagens claras na concisão e expressividade do código. Com isso, o Java deixou de ser estritamente orientado a objetos para se tornar uma linguagem de múltiplos paradigmas [4][7].

Entre os principais recursos adicionados, destacam-se as expressões lambda, as interfaces funcionais, a API de Streams, o tipo Optional e as referências a métodos. Esses elementos passaram a permitir que comportamentos fossem tratados como valores, promovendo um estilo mais declarativo, conciso e de alto nível de abstração [3].

As expressões lambda, por exemplo, possibilitam que comportamentos sejam passados como argumentos, facilitando a criação de APIs mais enxutas e flexíveis. Já a API de Streams viabiliza o processamento funcional de coleções, com suporte a operações encadeadas e paralelismo automático. O uso do tipo Optional, por sua vez, contribui para a redução de exceções relacionadas a valores nulos, como o conhecido NullPointerException [2][3].

Essas inovações impactam não apenas o modo de escrever código, mas também o

desenvolvimento de frameworks e bibliotecas. Tecnologias amplamente utilizadas no ecossistema Java como Spring, banco de dados Cassandra, implementação do Elastic Search e do Apache Kafka, passaram a incorporar os novos recursos - os quais serão mostrados alguns exemplos no subtópico 4.1.

No entanto, a adoção desses recursos não ocorre de maneira uniforme. Muitos projetos, especialmente no setor corporativo, continuam presos a versões anteriores do Java por restrições de compatibilidade e infraestrutura. Além disso, há uma curva de aprendizado significativa para equipes acostumadas ao estilo tradicional da linguagem, o que pode dificultar a transição para os novos padrões [7].

Diante desse cenário, compreender a evolução do Java a partir da versão 8 não se trata apenas de uma atualização técnica, mas de um passo estratégico para quem deseja manter a relevância de seus sistemas e melhorar a qualidade do software.

1.2 Objetivos

Este trabalho investiga os impactos e benefícios da refatoração e da adoção das funcionalidades introduzidas a partir da versão 8 do Java, com foco em projetos open source e empresariais.

Os objetivos específicos são:

I. Compreender a evolução da linguagem Java a partir da versão 8, destacando os principais recursos funcionais incorporados.

II. Analisar como esses recursos podem ser utilizados para refatorar código legado, avaliando os ganhos em legibilidade, desempenho e manutenção.

III. Estudar a adoção prática dessas funcionalidades em projetos reais, identificando padrões de uso, desafios enfrentados e estratégias de migração via entrevista por formulário com desenvolvedores Java iniciantes a profissionais.

IV. Apresentar evidências com base em repositórios de código aberto e contextos corporativos, a fim de ilustrar os efeitos concretos da transição para os novos recursos da linguagem.

Ao final deste estudo, espera-se contribuir para o entendimento das transformações ocorridas na linguagem Java com a chegada da versão 8, e oferecer subsídios para decisões técnicas relacionadas à modernização de código e adoção de boas práticas em projetos de

software.

1.3 Estrutura do Trabalho

Este trabalho está estruturado em seis capítulos, organizados de forma a conduzir o leitor desde a motivação inicial até as conclusões finais, conforme descrito a seguir:

- Capítulo 1 – Introdução: Apresenta o contexto da pesquisa, a motivação para a escolha do tema e os objetivos que nortearam o desenvolvimento deste estudo, situando a importância da modernização da linguagem Java no cenário contemporâneo da engenharia de software.
- Capítulo 2 – Fundamentos do Java 8+: Discute detalhadamente os principais recursos introduzidos a partir da versão 8 do Java, com destaque para expressões lambda, interfaces funcionais, Stream API, Optional e Method References. Cada funcionalidade é explicada em profundidade, com exemplos práticos ilustrando sua aplicação e relevância no contexto da refatoração de código.
- Capítulo 3 – Refatoração com Java 8+: Trata da aplicação prática dos recursos mencionados em códigos legados. São discutidas as motivações para refatorar, os benefícios observados e os desafios enfrentados. Trabalhos são utilizados para evidenciar o impacto concreto dessas transformações, bem como motivações e benefícios, assim como desafios e limitações.
- Capítulo 4 – Adoção em Diferentes Contextos: Analisa o processo de adoção das novas funcionalidades em projetos open source e ambientes corporativos, destacando fatores técnicos e as consequências da utilização das novas features. Todo o processo é exemplificado com trechos de códigos reais Open Source e Empresariais.
- Capítulo 5 – Experiência Prática e Percepção dos Desenvolvedores: Apresenta uma análise empírica realizada por meio de um formulário aplicado a desenvolvedores de diferentes instituições — como Caixa Econômica Federal, Banco do Brasil, além de discentes e profissionais participantes de projetos de inovação do Centro de Informática da Universidade Federal de Pernambuco (UFPE) —, trazendo dados estatísticos e análises qualitativas sobre a experiência com a adoção do Java 8+.
- Capítulo 6 – Conclusão: Reúne as reflexões finais do trabalho, sistematizando os principais achados, reconhecendo as limitações da pesquisa e propondo caminhos

para estudos futuros relacionados à modernização de sistemas com base nos recursos oferecidos pelas versões mais recentes da linguagem Java.

2. FUNDAMENTOS

2.1 Interfaces Funcionais

As interfaces funcionais, definidas por possuírem exatamente um único método abstrato, exercem um papel central na utilização das expressões lambda em Java. Elas funcionam como contratos que estabelecem o comportamento esperado, o qual será posteriormente definido de forma dinâmica por meio de lambdas [9].

Com a chegada do Java 8, foi introduzido o pacote **`java.util.function`**, que disponibiliza uma série de interfaces funcionais padronizadas, como Predicate, Function, Consumer e Supplier. Essas interfaces permitem a construção de comportamentos mais complexos a partir de componentes simples, reutilizáveis e de fácil teste, promovendo, assim, maior clareza e organização no desenvolvimento do código [9].

Esse novo modelo de programação possibilita uma integração mais harmônica entre o paradigma orientado a objetos e os princípios da programação funcional. Dessa forma, oferece ao desenvolvedor recursos para estruturar soluções de maneira mais declarativa, concisa e elegante.

A biblioteca padrão do Java passou a incluir diversas interfaces funcionais reutilizáveis no pacote java.util.function [9]. As mais comuns são:

- **`Predicate<T>`** – recebe um parâmetro do tipo T e retorna um valor booleano;
- **`Function<T, R>`** – recebe um parâmetro de tipo T e retorna um valor de tipo R;
- **`Consumer<T>`** – recebe um parâmetro de tipo T e não retorna nada;
- **`Supplier<T>`** – não recebe nenhum parâmetro e retorna um valor de tipo T.

Essa padronização permitiu não apenas a criação de código mais limpo e modular, mas também facilitou o uso de APIs genéricas e a reutilização de lógica, sem a necessidade de criar interfaces personalizadas para cada novo comportamento. Logo abaixo é demonstrado alguns exemplos.

Exemplo 1: Uso de Predicate

```
Predicate<String> isNotEmpty = s -> !s.isEmpty();

System.out.println(isNotEmpty.test("Java"));  // true
System.out.println(isNotEmpty.test(""));      // false

//Figura 2.1.1 código exemplificativo do uso de Predicate.
```

Figura 2.1.1 código exemplificativo do uso de Predicate.

Aqui é utilizado uma interface Predicate, a qual possui o método **`test`** que avalia a condição que foi passada para verificar se uma string não está vazia. A função lambda **`s → !s.isEmpty()`** define o comportamento esperado e pode ser passada como argumento para outros métodos que esperam um Predicate.

Exemplo 2: Uso de Function

```
Function<String, Integer> stringLength = s -> s.length();

System.out.println(stringLength.apply("Interface"));  // 9
```

Figura 2.1.2 código exemplificativo do uso de Function.

Esta função recebe uma string e retorna seu comprimento. O uso de **`Function<T, R>`** é útil em transformações de dados, especialmente em pipelines de Stream . Seguindo o mesmo raciocínio do exemplo anterior, há um método na Interface Funcional chamado "apply".

Exemplo 3: Uso de Consumer

```
Consumer<String> printUpperCase = s -> System.out.println(s.toUpperCase());

printUpperCase.accept("java");  // JAVA
```

Figura 2.1.3 código exemplificativo do uso de Consumer.

O Consumer é ideal para executar operações com efeitos colaterais, como imprimir na tela ou gravar logs.

Exemplo 4: Uso de Supplier

```
Supplier<LocalDate> currentDate = () -> LocalDate.now();

System.out.println(currentDate.get());  // imprime a data atual
```

Figura 2.1.4 código exemplificativo do uso de Supplier.

O Supplier não recebe nenhum parâmetro e apenas fornece valores. É bastante útil quando se deseja encapsular lógica de fornecimento de dados, como gerar identificadores ou datas.

2.2 Lambda Functions

A introdução das expressões lambda no Java 8 representou uma das transformações mais relevantes no paradigma de programação da linguagem. Essas expressões consistem em funções anônimas — blocos de código definidos de forma concisa — que podem ser tratados como objetos e passados como argumentos para métodos. Com uma sintaxe mais enxuta e expressiva, as expressões lambda substituem estruturas anteriormente mais complexas, como classes anônimas, proporcionando maior simplicidade e clareza ao código [3].

Entre os principais benefícios desse recurso, destaca-se a redução significativa do chamado boilerplate code — trechos repetitivos e detalhados que pouco contribuem para a lógica central da aplicação. O uso de expressões lambda é especialmente eficaz em operações com coleções, nas quais a intenção do programador se torna mais evidente e direta, favorecendo a legibilidade e a manutenção do código [3].

Além de favorecer a concisão, estudos apontam que as expressões lambda contribuem para o aumento da suncitez dos módulos do sistema, facilitando a refatoração de aplicações legadas [5]. Isso conduz a soluções mais funcionais, organizadas e alinhadas com boas práticas de desenvolvimento moderno.

Abaixo são listados alguns benefícios práticos através de exemplos:

Exemplo 1: Substituição de classe anônima por lambda

Antes do Java 8, era comum utilizar classes anônimas para instanciar interfaces funcionais:

```
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        System.out.println("Executando uma tarefa");
    }
};
```

Figura 2.2.1 classe anônima.

Com a introdução das expressões lambda, esse código pode ser simplificado de forma significativa:

```
Runnable runnable = () -> System.out.println("Executando uma tarefa");
```

Figura 2.2.2 substituição da classe anônima por lambda function.

Neste exemplo, a estrutura mais verbosa da classe anônima é substituída por uma única linha, mantendo o mesmo comportamento, porém com maior clareza.

Exemplo 2: Uso de lambda com List.forEach()

A API de coleções do Java também se beneficiou do uso de lambdas. Com o método forEach, é possível iterar sobre listas de forma mais expressiva:

```
List<String> nomes = Arrays.asList("Ana", "Bruno", "Carlos");

nomes.forEach(nome -> System.out.println(nome));
```

Figura 2.2.3 Lambda em forEach.

A expressão lambda **`nome → System.out.println(nome)`** define o comportamento aplicado a cada elemento da lista. A mesma operação, antes do Java 8, exigiria o uso de um laço “for” ou um Iterator, com mais linhas e menor legibilidade.

2.3 Streams API

A introdução da API de Streams no Java 8 representou um avanço significativo ao incorporar um modelo funcional para o processamento de coleções. Essa abordagem possibilita que sequências de dados sejam manipuladas de forma declarativa, dispensando a utilização de estruturas de repetição tradicionais, como os laços for e while. Com isso, o código torna-se substancialmente mais legível e alinhado com os princípios de programação moderna [2].

Os streams operam por meio de um encadeamento fluente de operações, permitindo que desenvolvedores combinem etapas intermediárias — como map, filter e sorted — com operações terminais — como collect, count e reduce [10] [9]. Essa composição encadeada viabiliza a construção de lógicas de transformação de dados de forma clara, sem a necessidade de um controle manual do fluxo de iteração, como ocorre nos modelos imperativos.

Outro aspecto fundamental da API de Streams é o mecanismo de avaliação preguiçosa (lazy evaluation). As operações intermediárias — como map, filter e distinct — não são executadas imediatamente quando declaradas, mas sim postergadas até que uma operação terminal, como collect ou count, seja invocada [10] [9]. Esse comportamento permite que o processamento seja otimizado, evitando cálculos desnecessários e melhorando o desempenho geral da aplicação. Dessa forma, apenas os elementos realmente necessários percorrem toda a cadeia de operações, o que contribui para uma execução mais eficiente e racional dos recursos computacionais.

Estudos recentes [10] demonstram que a API de Streams tem sido amplamente adotada em projetos de código aberto e por essa causa, tem despertado a atenção dos pesquisadores para essa “feature”. A facilidade de expressar operações complexas de forma concisa e legível reforça seu papel como um dos principais avanços da linguagem a partir do Java 8 [4]. Abaixo segue alguns exemplos:

Exemplo 1: Filtrar e mapear uma lista de nomes

```java
List<String> nomes = Arrays.asList("Ana", "Bruno", "Carlos", "Amanda", "Beatriz");

List<String> nomesComA = nomes.stream()
    .filter(nome -> nome.startsWith("A"))
    .map(nome -> nome.toUpperCase())
    .collect(Collectors.toList());

System.out.println(nomesComA); // [ANA, AMANDA]
```

Figura 2.3.1 filtro e mapa do stream.

Neste exemplo, são selecionados apenas os nomes que se iniciam com a letra "A", utilizando filter. Em seguida, com map, esses nomes são convertidos para letras maiúsculas. O resultado final é obtido por meio do método collect, que agrega os elementos em uma nova lista.

Exemplo 2: Somando valores pares

```java
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6);

int somaPares = numeros.stream()
    .filter(n -> n % 2 == 0)
    .reduce(0, (a, b) -> a + b);

System.out.println(somaPares); // 12
```

Figura 2.3.2 reduce do stream.

Neste caso, o uso de filter permite isolar os números pares da lista. Em seguida, o reduce é aplicado para somá-los. Essa operação terminal é muito útil quando se deseja condensar os elementos da stream em um único resultado, como uma soma, multiplicação ou mesmo uma string concatenada.

Exemplo 3: Removendo duplicatas e ordenando elementos

```java
List<Integer> numeros = Arrays.asList(3, 6, 3, 9, 6, 12, 15);

List<Integer> distintosOrdenados = numeros.stream()
    .distinct()
    .sorted()
    .collect(Collectors.toList());

System.out.println(distintosOrdenados); // [3, 6, 9, 12, 15]
```

Figura 2.3.3 distinct e sorted do stream.

Aqui o distinct é utilizado para eliminar elementos duplicados, enquanto sorted organiza os valores em ordem crescente. Essa combinação é particularmente útil em tarefas de pré-processamento de dados, onde é necessário limpar e ordenar coleções.

2.4 Optional

A introdução da classe **`Optional<T>`** no Java 8 teve como propósito principal oferecer uma alternativa mais segura e expressiva ao uso de valores nulos, frequentemente responsáveis por falhas em tempo de execução, como o conhecido NullPointerException [6]. Em vez de retornar diretamente um valor que pode ser nulo, os métodos passaram a ter a possibilidade de retornar um objeto do tipo Optional, o qual representa de maneira explícita a presença ou a ausência de um valor.

Esse recurso incentiva o desenvolvedor a lidar de forma mais consciente com cenários em que os dados podem estar ausentes, por meio de métodos específicos como **`isPresent()`**, **`ifPresent()`**, **`orElse()`** e **`orElseThrow()`** [9]. Ao adotar esse modelo, promove-se um fluxo de controle mais robusto, transparente e menos propenso a falhas silenciosas.

Abaixo estão alguns exemplos práticos de uso da classe **`Optional<T>`**, cada um seguido de uma explicação clara:

Exemplo 1: Retorno de Optional em métodos de busca

```java
public Optional<String> buscarNomePorId(int id) {
    if (id == 1) {
        return Optional.of("João");
    } else {
        return Optional.empty();
    }
}
```

Figura 2.4.1 `Optional.of()` e `Optional.empty()`.

Neste exemplo, o método buscarNomePorId retorna um objeto do tipo **`Optional<String>`** com base no valor do identificador fornecido. Caso o identificador seja igual a 1, considera-se que o nome foi encontrado e retorna-se um Optional preenchido com esse valor. Em qualquer outra situação, retorna-se um **`Optional.empty()`**, o que indica, de forma explícita, a ausência de resultado. Tal prática evita o uso de null e favorece a clareza na manipulação dos retornos.

Exemplo 2: Execução condicional com ifPresent()

```java
Optional<String> nome = Optional.of("Maria");

nome.ifPresent(n -> System.out.println("Nome encontrado: " + n));
```

Figura 2.4.2 `ifPresent()` do Optional.

Neste caso, assume-se que o valor está presente, e utiliza-se o método **`ifPresent()`** para executar uma ação condicional. Caso o Optional contenha um valor, a função lambda será invocada para exibir a mensagem no console. Essa abordagem elimina a necessidade de verificações explícitas com **`if`** e **`null`**, promovendo um código mais enxuto e seguro.

Exemplo 3: Definição de valor padrão com orElse()

```
Optional<String> nome = Optional.empty();

String resultado = nome.orElse("Nome padrão");

System.out.println(resultado);
```

Figura 2.4.3 `orElse()` do Optional.

Neste exemplo, o Optional não possui valor. Utiliza-se, então, o método **`orElse()`** para fornecer um valor padrão, que será utilizado na ausência de conteúdo. Esse padrão é bastante útil quando há uma alternativa viável a ser adotada, evitando, assim, falhas inesperadas no fluxo da aplicação.

Exemplo 4: Tratamento de exceção com orElseThrow()

```
Optional<String> nome = Optional.empty();

String resultado = nome.orElseThrow(() -> new IllegalArgumentException("Valor não encontrado"));
```

Figura 2.4.4 `orElseThrow()` do Optional.

Aqui, o valor ausente no Optional é tratado de forma rigorosa. Com o uso do método **`orElseThrow()`**, lança-se uma exceção personalizada caso o valor esperado não esteja presente. Tal abordagem é recomendada quando a ausência de dados representa uma violação de regra de negócio ou um erro que deve ser explicitamente tratado pela aplicação.

2.5 Method References

As referências a métodos (Method References) representam uma alternativa ainda mais concisa e elegante ao uso de expressões lambda, especialmente nos casos em que a função anônima se limita à chamada de um método já existente. Por meio desse recurso, introduzido a partir do Java 8, a linguagem passou a permitir que métodos estáticos, de instância ou até mesmo construtores fossem diretamente utilizados como argumentos de funções, conferindo maior clareza ao código [11][12].

Há, essencialmente, quatro categorias de referências a métodos:

- Referências a métodos estáticos (Classe::nomeDoMetodo),
- Referências a métodos de instância de um objeto específico (objeto::nomeDoMetodo),
- Referências a métodos de instância de um objeto arbitrário de um determinado tipo (Classe::nomeDoMetodo), e
- Referências a construtores (Classe::new).

O uso apropriado dessas referências contribui significativamente para a legibilidade e manutenção do código, além de favorecer a construção de APIs mais expressivas, reutilizáveis e em sintonia com os princípios da programação funcional [11][12]. Trata-se, portanto, de um recurso que promove um estilo de desenvolvimento mais moderno e alinhado com as boas práticas adotadas em ambientes de software robustos e escaláveis.

Abaixo segue um exemplo para cada categoria:

Exemplo 1: Referência a métodos estáticos (Classe::nomeDoMetodo)

```java
import java.util.Arrays;
import java.util.List;

public class ExemploMetodoEstatico {
    public static void imprimir(String mensagem) {
        System.out.println(mensagem);
    }

    public static void main(String[] args) {
        List<String> mensagens = Arrays.asList("Olá", "Mundo", "Java 8");

        mensagens.forEach(ExemploMetodoEstatico::imprimir);
    }
}
```

Figura 2.5.1 Métodos estáticos.

Neste exemplo, o método estático imprimir é passado como referência para o forEach, que o invoca para cada elemento da lista. O uso de ExemploMetodoEstatico::imprimir torna o código mais limpo e direto, substituindo uma expressão lambda como **`s → imprimir(s)`**.

Exemplo 2: Referência a métodos de instância de um objeto específico (objeto::nomeDoMetodo)

```java
import java.util.Arrays;
import java.util.List;

public class ExemploInstancia {
    public void exibir(String texto) {
        System.out.println(">> " + texto);
    }

    public static void main(String[] args) {
        ExemploInstancia exemplo = new ExemploInstancia();
        List<String> nomes = Arrays.asList("Alice", "Bruno", "Carlos");

        nomes.forEach(exemplo::exibir);
    }
}
```

Figura 2.5.2 Métodos de instância de um objeto específico.

Aqui, a referência ao método de instância exibir é feita a partir de um objeto específico da classe ExemploInstancia, substituindo a expressão **`x→exemplo.exibir(x)`** por exemplo. Cada elemento da lista é passado como argumento para o método, que o imprime com um prefixo.

Exemplo 3: Referência a métodos de instância de um objeto arbitrário de um tipo (Classe::nomeDoMetodo)

```java
import java.util.Arrays;
import java.util.List;

public class ExemploArbitrario {
    public static void main(String[] args) {
        List<String> palavras = Arrays.asList("java", "lambda", "stream");

        palavras.sort(String::compareToIgnoreCase);

        palavras.forEach(System.out::println);
    }
}
```

Figura 2.5.3 Métodos de instância de um objeto arbitrário.

Neste caso, a referência é feita a um método de instância (compareToIgnoreCase) da classe String, aplicado a cada par de elementos da lista durante a ordenação. O compilador entende que o primeiro elemento é o receptor da chamada e o segundo é o argumento do método. Sem o method reference, seria **`palavras.sort((s1, s2)→s1.compareToIgnoreCase(s2))`**.

Exemplo 4: Referência a construtores (Classe::new)

```java
import java.util.function.Supplier;

public class ExemploConstrutor {
    static class Pessoa {
        public Pessoa() {
            System.out.println("Pessoa criada!");
        }
    }

    public static void main(String[] args) {
        Supplier<Pessoa> criador = Pessoa::new;

        Pessoa p = criador.get();  // Instancia uma nova Pessoa
    }
}
```

Figura 2.5.4 Métodos de instância de um objeto arbitrário.

Aqui, a referência Pessoa::new é utilizada para criar uma instância da classe Pessoa. O Supplier é uma interface funcional que representa uma função sem parâmetros que retorna um valor (como mencionado no subtópico 2.1)— neste caso, um novo objeto Pessoa.

Apenas a fim de mostrar a correlação entre os assuntos, segue a mesmas figuras 2.3.1 e 2.3.2 utilizadas para exemplificar a api *streams*, porém utilizando “Method Reference” para deixar o código mais enxuto:

```java
List<String> nomes = Arrays.asList("Ana", "Bruno", "Carlos", "Amanda", "Beatriz");

List<String> nomesComA = nomes.stream()
    .filter(nome -> nome.startsWith("A"))
    .map(String::toUpperCase)
    .collect(Collectors.toList());

System.out.println(nomesComA); // [ANA, AMANDA]
```

Figura 2.5.5 filter com method reference.

Neste código, o trecho **`nome → nome.toUpperCase()`** foi substituído por "String::toUpperCase", uma vez que pode aplicar o exemplo 3 aqui, já que sempre a partir do argumento da função lambda no argumento do map sempre é aplicado a função toUpperCase() da instância da Classe String.

```java
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6);

int somaPares = numeros.stream()
    .filter(n -> n % 2 == 0)
    .reduce(0, Integer::sum);

System.out.println(somaPares); // 12
```

Figura 2.5.6 reduce com method reference.

Já neste caso, a expressão **`(a, b) → a + b`** por ser trocada por **`Integer::sum`**, ou seja é aplicado o Exemplo 1, pois o método sum é estático da classe Integer. Como na Classe Integer há o método "public static int sum(int a, int b)" e possui a mesma assinatura e lógica da expressão anterior, então pode substituir um pelo outro.

3. REFATORAÇÃO

3.1 Refatoração de Código Legado com Java 8+

A prática da refatoração consiste na reorganização interna do código-fonte com o objetivo de melhorar sua estrutura e qualidade, sem modificar o comportamento funcional da aplicação. Com o advento das funcionalidades introduzidas a partir do Java 8, como expressões lambda, interfaces funcionais, a API de Streams, o uso da classe Optional e referências a métodos, ampliaram-se significativamente as possibilidades de reescrita de trechos legados de forma mais expressiva, modular e eficiente [3].

Códigos desenvolvidos em versões anteriores da linguagem frequentemente apresentam construções imperativas e repetitivas, cuja manutenção pode se tornar onerosa com o tempo. A refatoração com recursos do Java 8 possibilita a transição para um modelo mais declarativo, eliminando a necessidade de estruturas como loops explícitos ou classes anônimas. A adoção dessas práticas resulta em maior legibilidade, redução de código redundante e melhoria na coesão dos módulos [3][8].

Contudo, é fundamental destacar que a refatoração de sistemas legados requer planejamento criterioso e profundo conhecimento do comportamento atual do software. A simples reescrita sintática, sem a adequada validação, pode acarretar regressões funcionais. Assim, testes automatizados desempenham papel crucial na preservação da integridade do sistema ao longo do processo de modernização, apesar de não fazer parte do escopo deste trabalho.

3.2 Motivações e Benefícios

A motivação principal para a refatoração com recursos funcionais do Java 8 reside na busca por um código mais limpo, expressivo e alinhado às boas práticas de engenharia de software. O uso de expressões lambda e da API de Streams, por exemplo, permite substituir estruturas excessivamente verbosas por construções concisas, promovendo a clareza e a expressividade do código [3].

Estudos empíricos têm demonstrado que a adoção dessas funcionalidades está relacionada a ganhos objetivos de qualidade. Entre os principais benefícios observados,

destacam-se a elevação da densidade funcional por linha de código, a redução da complexidade de leitura de código e o aumento da facilidade de escrita de código paralelo [4]. Esses fatores favorecem a manutenção e a evolução contínua do sistema, além de reduzir o risco de introdução de erros.

Outro aspecto relevante é a melhora na reutilização de código. Interfaces funcionais, associadas a expressões lambda, permitem o encapsulamento de comportamentos em unidades reaproveitáveis, facilitando a composição e a testabilidade. Assim, desenvolvedores conseguem implementar soluções mais robustas e de fácil adaptação frente a mudanças de requisitos.

3.3 Desafios e Limitações

Apesar das vantagens amplamente reconhecidas, a refatoração de código com recursos do Java da versão 8 em diante também impõe desafios que não podem ser ignorados. Um dos principais obstáculos é a necessidade de mudança de paradigma por parte dos desenvolvedores, muitos dos quais estão habituados a estilos imperativos de programação. A introdução de conceitos como imutabilidade, funções puras (não possuem efeitos colaterais, sempre produz a mesma saída com a entrada igual) e composição funcional exige capacitação técnica e tempo de adaptação [4][3].

Além disso, nem todos os contextos de desenvolvimento estão preparados para a plena adoção das novas funcionalidades. Projetos que precisam manter compatibilidade com versões anteriores do Java enfrentam restrições que podem inviabilizar, total ou parcialmente, a refatoração funcional [7]. Em tais cenários, é necessário adotar estratégias híbridas, conciliando novas práticas com estruturas legadas.

Outra limitação recorrente diz respeito ao desempenho. Embora a API de Streams traga ganhos de legibilidade e clareza, seu uso inadequado pode acarretar sobrecarga computacional, especialmente em operações sobre grandes volumes de dados. A criação de pipelines longos e a geração excessiva de objetos intermediários são exemplos de práticas que, sem a devida análise, podem comprometer a eficiência do sistema [2].

Por fim, é imperativo que toda refatoração funcional seja acompanhada de uma cobertura de testes adequada, garantindo que a nova implementação preserve integralmente a lógica de negócio original. O sucesso da modernização depende, em grande medida, da

capacidade de assegurar que as transformações sejam corretas e seguras.

4. ANÁLISE EM PROJETOS REAIS: ADOÇÃO DAS FUNCIONALIDADES DO JAVA 8+

Este capítulo apresenta uma análise prática da adoção dos recursos introduzidos a partir do Java 8, considerando tanto projetos de código aberto quanto sistemas desenvolvidos no contexto empresarial. O objetivo é ilustrar, de maneira concreta, os efeitos da transição para essas novas funcionalidades, destacando os ganhos relacionados à legibilidade, desempenho e manutenção do código.

Ademais, examina-se como expressões lambda, interfaces funcionais, a API de Streams, a classe Optional, funções lambda e referências a métodos podem ser aplicados no processo de modernização de bases de código legado, proporcionando maior eficiência e qualidade no desenvolvimento de software.

4.1 Aplicação das Funcionalidades em Projetos Open Source

A adoção das funcionalidades introduzidas a partir do Java 8, como Streams, Lambdas, Optional e Method References, transformou a maneira como projetos open source estruturam e organizam seu código. Estas mudanças resultaram em melhorias expressivas na legibilidade, concisão e manutenção dos sistemas, ao permitir a modelagem de operações de forma mais declarativa e funcional [2][9].

A seguir, apresentam-se exemplos retirados diretamente de repositórios públicos, com uma análise minuciosa dos impactos e benefícios obtidos por meio da adoção desses recursos.

Exemplo 1: Utilização da API de Streams com Predicados Compostos na Busca de Anotações

```java
public static @Nullable Class<?> findAnnotationDeclaringClassForTypes(
        List<Class<? extends Annotation>> annotationTypes, @Nullable Class<?> clazz) {

    if (clazz == null) {
        return null;
    }

    MergedAnnotation<?> merged = MergedAnnotations.from(clazz, SearchStrategy.SUPERCLASS)
   .stream().filter(MergedAnnotationPredicates.typeIn(annotationTypes)
   .and(MergedAnnotation::isDirectlyPresent)).findFirst().orElse(null);

    return (merged != null && merged.getSource() instanceof Class<?> sourceClass
            ? sourceClass : null);
}
```

Fonte: https://github.com/spring-projects/spring-framework/blob/main/spring-core/src/main/java/org/springframework/core/annotation/AnnotationUtils.java spring-projects/spring-framework – AnnotationUtils.java (linha 637) - Data de acesso: 26 de abril de 2025.

Neste exemplo, o uso da API de Streams é fundamental para percorrer anotações de uma classe, filtrá-las com base em um conjunto de critérios compostos (tipos e presença direta) e obter o primeiro resultado encontrado. Sem o uso do Stream, seria necessário construir um laço for, criar variáveis auxiliares e múltiplas estruturas de controle.

A combinação de filter com predicados compostos torna a lógica extremamente legível e o propósito da função se torna autoexplicativo. Além disso, o uso do Optional **`orElse(null)`** protege contra acessos a elementos inexistentes de maneira elegante e segura. Isso reduz a quantidade de código necessário, aumenta a robustez contra erros e favorece a manutenção futura, já que a modificação de critérios de busca, por exemplo, pode ser feita de maneira isolada e funcional.

Exemplo 2: Validação Funcional com Stream e Referência a Método

```
private SearchProfileResults buildSearchProfileResults(
    Collection<? extends SearchPhaseResult> fetchResults) {

    if (profileBuilder == null) {
        assert fetchResults.stream().map(SearchPhaseResult::fetchResult)
                                    .allMatch(r -> r == null || r.profileResult() == null)
                                    : "found fetch profile without search profile";
        return null;
    }
    return profileBuilder.build(fetchResults);
}
```

Fonte: https://github.com/elastic/elasticsearch/blob/main/server/src/main/java/org/elasticsearch/action/search/SearchPhaseController.java elastic/elasticsearch – SearchPhaseController.java (linha 645) - Data de acesso: 26 de abril de 2025.

Este trecho de código demonstra uma validação assertiva utilizando Streams combinados a referência a métodos (SearchPhaseResult::fetchResult). Em vez de percorrer a coleção com um for manual e aplicar condições sobre cada item, o desenvolvedor expressa toda a intenção lógica de forma encadeada.

A clareza de `stream().map(...).allMatch(...)` revela imediatamente que se está validando todos os resultados, simplificando drasticamente o entendimento e a revisão do código. Além disso, por ser uma construção funcional, o código é potencialmente mais otimizado para execução, interrompendo o processamento na primeira violação da condição (curto-circuito, operação que não precisa iterar por toda lista). A manutenção futura também é facilitada, uma vez que a modificação ou adição de novas condições seria incorporada diretamente no pipeline funcional.

Exemplo 3: Filtragem Elegante de Nulos com API de Streams

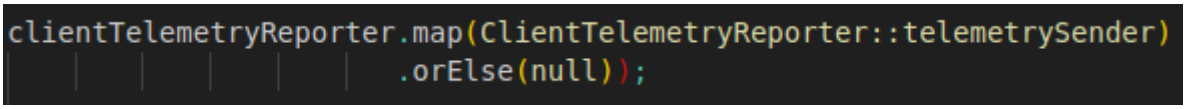
```
List<TopDocs> topDocsList = results
    .stream().filter(Objects::nonNull).toList();
```

Fonte: https://github.com/elastic/elasticsearch/blob/main/server/src/main/java/org/elasticsearch/action/search/SearchPhaseController.java elastic/elasticsearch – SearchPhaseController.java (linha 144) - Data de acesso: 26 de abril de 2025.

Aqui observa-se um exemplo clássico de como a API de Streams melhora a manipulação de coleções. Antes do Java 8, seria necessário criar uma lista manualmente, iterar sobre results com um for, testar cada elemento e adicionar os não nulos. Agora, em apenas duas linhas, o mesmo resultado é obtido de forma extremamente concisa e declarativa.

O uso de **`Objects::nonNull`** também evita erros triviais (como comparações em um Predicate - subtópico 2.1) e reforça a intenção de manter apenas os elementos válidos na coleção. Em relação à legibilidade, o desenvolvedor compreende o que está acontecendo sem precisar interpretar várias instruções dispersas.

Exemplo 4: Tratamento Seguro de Possíveis Ausências com Optional



Fonte: https://github.com/apache/kafka/blob/trunk/clients/src/main/java/org/apache/kafka/clients/producer/KafkaProducer.java Kafka - KafkaProducer.java (linha 524) - Data de acesso: 26 de abril de 2025.

O código ilustra a utilização moderna do Optional, evitando verificações explícitas de null que eram fonte comum de erros no Java tradicional. Em vez de um **`if (clientTelemetryReporter != null) {...}`**, o fluxo de tratamento é embutido diretamente na cadeia de chamadas.

A função map transforma o valor existente e **`orElse()`** fornece um fallback seguro. Essa abordagem não só torna o código mais robusto contra exceções inesperadas, como também melhora a clareza do fluxo de dados. No aspecto de manutenção, quaisquer alterações no comportamento de fallback podem ser feitas facilmente sem reestruturar blocos condicionais complexos.

Exemplo 5: Construção de Pipeline Completo de Dados com API de Streams

```java
public List<String> getAccordManagedKeyspaces()
{
    Keyspaces keyspaces = Schema.instance.getNonLocalStrategyKeyspaces();
    return keyspaces.stream().flatMap(ks -> ks.tables.stream())
                .filter(TableMetadata::requiresAccordSupport)
                .map(tbm -> tbm.keyspace)
                .distinct()
                .sorted()
                .collect(toList());
}
```

Fonte:

https://github.com/apache/cassandra/blob/trunk/src/java/org/apache/cassandra/service/StorageService.java

Cassandra - StorageService.java (linha 4226) -

Data de acesso: 26 de abril de 2025.

Este exemplo ilustra um pipeline complexo de transformação de dados, realizado de maneira fluente e altamente legível.

Sem a API de Streams, seria necessário percorrer manualmente as keyspaces, iterar sobre as tabelas, aplicar filtros, construir uma lista sem duplicações, ordenar manualmente e finalmente coletar os dados — tudo isso disperso em múltiplos blocos de código.

Graças ao encadeamento de flatMap, filter, map, distinct, sorted e collect, toda a transformação é descrita em um fluxo linear e natural, evidenciando passo a passo a transformação de dados. Essa abordagem funcional reduz drasticamente a complexidade e o tamanho do código, além de ser facilmente modificável e extensível.

4.2 Avaliação das Funcionalidades em Projetos Open Source

A análise de trechos de código provenientes de projetos open source de ampla relevância permitiu constatar, de maneira concreta, os impactos positivos advindos da adoção dos recursos introduzidos a partir do Java 8. O uso de expressões lambda, Streams API, Optional e referências a métodos revelou-se fundamental para a evolução da qualidade do código, refletindo-se em ganhos expressivos de legibilidade, manutenção e, em determinados contextos, desempenho.

Observa-se que a transição de estruturas tradicionais, como laços de repetição e verificações manuais de nulidade, para abordagens funcionais e declarativas, reduziu

consideravelmente a quantidade de código boilerplate, tornando a intenção do programador mais explícita e o fluxo de dados mais linear e compreensível. Essa melhoria na clareza do código favorece a colaboração entre equipes de desenvolvimento, a incorporação de novos membros aos projetos e a manutenção de longo prazo.

No tocante à manutenção, a modularização proporcionada pelas novas funcionalidades facilita a extensão e a modificação de sistemas, uma vez que cada etapa da transformação de dados é delineada de forma isolada e encadeada. Entretanto, a debugabilidade pode ser prejudicada, já que há menos locais no código para inserção de breakpoints.

Embora os ganhos de desempenho dependam do contexto de uso, a Streams API, especialmente quando combinada a operações de curto-circuito e a estratégias adequadas de processamento, possibilita otimizações internas que tendem a beneficiar a eficiência da aplicação sem demandar esforços adicionais do desenvolvedor.

Já a utilização de Optional, por sua vez, reforça práticas de programação mais seguras, ao eliminar, de forma sistemática, a vulnerabilidade a exceções relacionadas a valores nulos.

Dessa forma, evidencia-se que a incorporação das funcionalidades do Java 8+ não constitui apenas uma modernização estética do código, mas sim uma transformação substancial na maneira como as soluções são projetadas e evoluídas. A experiência prática verificada em projetos consolidados reforça a relevância da refatoração de sistemas legados, apontando para um caminho de maior sustentabilidade, robustez e manutenibilidade no desenvolvimento de software contemporâneo.

4.3 Aplicação das Funcionalidades em Projetos Empresariais

No ambiente empresarial atual, a modernização tecnológica tornou-se uma necessidade estratégica. Refatorar aplicações legadas, utilizando os recursos introduzidos a partir do Java 8, tem desempenhado um papel central nesse processo de transformação. Essa evolução não apenas atualiza a infraestrutura tecnológica das empresas, mas também traz ganhos reais na produtividade das equipes, na qualidade do código e na manutenção de sistemas cada vez mais complexos [7].

A adoção de expressões lambda, da API de Streams, da classe Optional e de referências a métodos em projetos corporativos não é amplamente adotada em projetos

empresariais [7], apesar de seus benefícios como a redução de código redundante, a melhora na clareza da lógica implementada e a diminuição da incidência de erros comuns (como o NullPointerException por exemplo) – seja por boa parte dos projetos corporativos manterem a compatibilidade com sistemas legados e ainda haver suporte para features antigas ao java 8+, seja por comodidade ou apredizado dos desenvolvedores.

Na sequência, serão apresentados exemplos práticos, proveniente de um projeto empresarial da Caixa Econômica Federal, onde a utilização dessas novas funcionalidades ilustra claramente os avanços alcançados. Todos os exemplos foram adaptados a fim de não comprometer o sigilo negocial da Empresa, ou seja, o intuito é apenas representar a lógica com as funcionalidades do java citadas neste documento, e não o domínio do negócio. Cada exemplo ilustra como as novas funcionalidades do Java 8, como expressões lambda, Streams, Optional e referências a métodos, têm sido aplicadas de forma eficaz no desenvolvimento e na refatoração de aplicações da empresa.

Exemplo 1: Construção de Entidades de Forma Segura e Concisa

```
final var entityBuilder = entity.builder().id(compositeId);

entrada.ifPresent(entityBuilder::entradaEntity);
tpAuth.ifPresent(entityBuilder::tpAuthEntity);
senha.ifPresent(entityBuilder::senhaEntity);
```

Neste primeiro exemplo, observa-se o uso sistemático da classe Optional e de referências a métodos para realizar o preenchimento condicional de atributos da entidade.

Antes da introdução do Optional, o padrão comum seria realizar verificações explícitas de nulidade **`if (obj != null)`** para cada campo, o que aumentava o tamanho do código e abria margem para erros como o famoso NullPointerException.

A utilização do ifPresent torna o fluxo mais enxuto, promovendo a separação clara entre o momento de verificação e a ação sobre o objeto.

Além disso, as referências diretas aos métodos do entityBuilder (**`entityBuilder::entradaEntity`** – exemplo 1 do subtópico 2.5) eliminam a necessidade de lambdas mais extensas (a qual seria "**`entrada→entityBuilder.entradaEntity(entrada)`**", por exemplo) ,

favorecendo a legibilidade.

Essa combinação resulta em um código mais expressivo e de fácil manutenção, alinhado às boas práticas de programação funcional, além de reduzir substancialmente o risco de defeitos relacionados a objetos nulos.

Exemplo 2: Conversão de Entidades com API de Streams

```
List<Entity> entities = lógica para retornar uma lista;

return entities.stream()
    .map(Converter::fromEntity)
    .toList();
```

Este exemplo ilustra a conversão de uma lista de entidades para objetos de outro domínio utilizando a API de Streams com uma referência direta a método estático da classe Converter (exemplo 1 do subtópico 2.5).

No modelo tradicional anterior ao Java 8, seria necessário iterar explicitamente sobre a coleção, instanciar manualmente objetos e adicioná-los a uma nova lista, resultando em um código prolixo e propenso a inconsistências.

O uso da operação map, associado à referência **`Converter::fromEntity`**, transforma cada elemento de maneira declarativa e homogênea, promovendo um padrão de codificação mais previsível e seguro.

A chamada a **`toList()`** complementa o processo, criando uma lista imutável em alguns contextos ou, no mínimo, indicando a intenção de apenas materializar o fluxo em memória.

Além de ganhos evidentes em clareza e concisão, o padrão adotado permite ganhos de desempenho futuros, uma vez que operações sobre Streams são otimizáveis e podem ser paralelizadas com relativa facilidade (o que não faz parte do escopo deste trabalho).

Exemplo 3: Atualização Condicional de Entidade

```
entity.ifPresent(
    presentEntity -> {
      presentEntity.setSituacao(SituacaoEnum.ATIVO);
      repository.persistAndFlush(presentEntity);
    });
```

Neste cenário, a atualização de uma entidade ocorre apenas se a mesma estiver efetivamente disponível, utilizando Optional para encapsular a lógica de verificação.

A abordagem tradicional exigiria o uso explícito de uma instrução condicional **`if (entity != null)`**, o que introduz ruído visual e aumenta o risco de omissão de verificações em trechos posteriores do código.

O encapsulamento da ação de persistência dentro do **`ifPresent()`** garante não apenas a segurança contra nulidade, mas também organiza o fluxo de forma mais linear e compreensível.

Esse padrão é especialmente vantajoso em sistemas críticos como os da Caixa Econômica Federal, onde falhas silenciosas em atualizações de estado podem gerar inconsistências graves no banco de dados.

Exemplo 4: Construção Dinâmica de Fragmento de Consulta

```
protected String buildQueryFilterFragment(
    List<Pair<String, Object>> filters, PrimitiveIterator.OfInt counter) {

    return filters.stream()
        .map(Pair::getLeft)
        .map(fieldName -> String.format("%s = ?%d", fieldName, counter.nextInt()))
        .takeWhile(i -> counter.hasNext())
        .reduce(
            new StringBuilder(String.valueOf(true)),
            (l, r) -> l.append(" AND ").append(r),
            StringBuilder::append)
        .toString();
}
```

Este exemplo explora o potencial da API de Streams para construir dinamicamente trechos de cláusulas SQL a partir de filtros fornecidos em tempo de execução.

Cada transformação é realizada de maneira declarativa: primeiro, extraem-se os

nomes dos campos (`map(Pair::getLeft)`), depois formata-se a cláusula para cada campo (String.format).

O uso de `PrimitiveIterator.OfInt` como contador garante a geração sequencial de parâmetros posicionais na query (?1, ?2, etc.), evitando o gerenciamento manual de índices, que seria mais suscetível a falhas.

O `takeWhile()` garante a condição de parada se `counter.hasNext()` for falso. Nesse trecho é admitido que sempre irá haver pelo menos 1 filtro na query do SQL.

A operação reduce, aqui utilizada para concatenar fragmentos, cria uma cadeia lógica de condições AND, favorecendo tanto a clareza quanto a escalabilidade da solução.

Essa estrutura não apenas simplifica o processo de manutenção e extensão de filtros, como também contribui para uma melhor gestão da segurança contra injeção de SQL, ao permitir que a consulta final mantenha a separação entre estrutura (vinda a partir do próprio código) e valores (garantindo a sanitização por meio da API do Java).

Exemplo 5: Identificação do Campo de ID com Stream e Optional

```
protected Optional<String> getIdFieldNameOptional() {
    return Arrays.stream(getEntityClass().getDeclaredFields())
        .filter(
            f ->
                f.isAnnotationPresent(Id.class)
                    || f.isAnnotationPresent(EmbeddedId.class)
                    || StringUtils.equalsIgnoreCase(f.getName(), "id"))
        .map(Field::getName)
        .findFirst();
}
```

A identificação dinâmica do campo de chave primária de uma entidade é um requisito recorrente em frameworks de persistência customizados.

Ao utilizar Streams e Optional, este método torna o processo de localização muito mais conciso e seguro.

As condições de filtragem são expressas de maneira direta, priorizando anotações padrão do JPA (`@Id` e `@EmbeddedId`) e fallback para nomes convencionais (`"id"`).

O uso de findFirst seguido de encapsulamento em Optional é particularmente interessante, pois obriga o consumidor deste método a lidar explicitamente com a

possibilidade de ausência de resultados, eliminando suposições inseguras.

Além disso, a abordagem baseada em Streams evita a construção manual de laços de repetição e reduz o risco de erros de lógica ao percorrer reflexivamente os atributos de uma classe.

Exemplo 6: Filtragem de Dados para Consultas

```
private String buildFiltersQuery(
    final List<Pair<String, @Nullable Object>> rawFilters,
    @Nullable String textSearchQuery) {

    final var filters =
        rawFilters.stream()
            .filter(f -> !Objects.isNull(f.getRight()))
            .filter(f -> !(f.getRight() instanceof CharSequence s)
                        || !StringUtils.isBlank(s))
            .toList();

    ...
}
```

Neste último exemplo, a filtragem prévia de critérios de busca inválidos é conduzida de forma funcional com Streams.

A primeira filtragem elimina pares cujo valor é nulo, enquanto a segunda assegura que, caso o valor seja uma string, ele não esteja vazio.

Essa dupla validação evita o envio de parâmetros desnecessários para o repositório de dados, reduzindo o risco de construção de consultas ineficientes ou incorretas.

O resultado é um código robusto, que se adapta facilmente a novos requisitos de validação, sem a necessidade de alterar o fluxo principal de construção da consulta.

Além disso, a clareza do pipeline de filtros facilita a auditoria e a compreensão de regras de negócio, fundamentais em ambientes bancários que requerem altíssimo nível de conformidade.

4.4 Avaliação das Funcionalidades em Projetos Empresariais

Os exemplos apresentados demonstram, de forma concreta, os benefícios práticos da

adoção das funcionalidades modernas do Java 8 em projetos empresariais.

A utilização de Streams, Optional, expressões lambda e referências a métodos contribui decisivamente para o aumento da legibilidade, da segurança e da eficiência do código.

Tais avanços não apenas aprimoram a qualidade técnica dos sistemas, como também oferecem uma base mais sólida para a manutenção e a expansão sustentável das aplicações no longo prazo.

Ademais, é perceptível que o uso adequado das novas funcionalidades do java 8 contribue na clareza do código, uma vez que ao invés de haver vários loops aninhados com várias regras de negócio da corporação, há apenas palavras (métodos de Streams) que possuem regras de negócios condensadas (Lambda Functions ou Method Reference, que por sua vez se utilizam de Interfaces Funcionais) com tratamento explícito de nulidade (Optional).

5. ADOÇÃO, EXPERIÊNCIA E CURVA DE APRENDIZADO

A evolução contínua da linguagem Java, em especial a partir da versão 8, tem provocado transformações significativas na forma como os desenvolvedores escrevem, refatoram e compreendem o código. Para investigar como as novas funcionalidades vêm sendo assimiladas por profissionais e estudantes da área de desenvolvimento, foi conduzida uma pesquisa com desenvolvedores de instituições públicas financeiras – como a Caixa Econômica Federal e o Banco do Brasil, além de discentes e profissionais participantes de projetos de inovação do Centro de Informática da Universidade Federal de Pernambuco (UFPE). Este capítulo apresenta uma análise dos resultados dessa pesquisa, com foco na adoção das funcionalidades do Java 8+, nas experiências práticas com refatoração e na percepção da curva de aprendizado associada a essas mudanças.

5.1 Perfil dos Participantes e Cenário de Adoção

A maioria dos participantes da pesquisa se identifica com um nível intermediário de experiência em Java (15 dos 22 respondentes), com menor presença de iniciantes (2) e desenvolvedores avançados (5). Esse cenário é compatível com os perfis profissionais das instituições mencionadas, refletindo um ambiente predominantemente corporativo.

Nível de Experiência em Java	Total
Intermediário (1 a 5 anos)	15
Avançado (mais de 5 anos)	5
Iniciante (menos de 1 ano)	2

Tabela 5.1.1 Experiência em Java

Em relação à atuação profissional, a maioria atua em desenvolvimento corporativo:

Área de Atuação	Total
Desenvolvimento de software empresarial	19
Desenvolvimento de software open source	2
Desenvolvimento com Visual Basic (contexto externo)	1

Tabela 5.1.2 Área de atuação

No que tange ao uso de versões do Java em ambientes produtivos, ainda que a maioria tenha adotado o Java 11 ou superior (11 respondentes), há um número relevante de contextos que ainda operam com o Java 8 (9), e até mesmo versões anteriores como o Java 6 (2). Isso reflete a realidade de manutenção de sistemas legados em instituições robustas.

5.2 Experiência com as Funcionalidades do Java 8+: Uso, Aprendizado e Desafios

A ampla maioria (95,5%) declarou já ter utilizado pelo menos uma das funcionalidades listadas neste trabalho (Streams API, Lambda Functions, Optional, Interfaces Funcionais e Method References) do Java 8+.

Entretanto, a adoção institucional, como através de treinamentos, guias de estilo e revisão de código, mostrou-se pouco presente. Apenas 3 participantes relataram incentivos formais, o que sugere uma predominância de aprendizado autodidata ou por meio de troca entre pares.

Adoção de Funcionalidades Java 8+	Total
Sim	21
Não	1

Tabela 5.2.1 Adoção das novas features

Houve incentivo institucional?	Total
Sim	3
Não	19

Tabela 5.2.2 Incentivo Institucional

Apesar da ausência de programas formais de capacitação, 14 dos 22 respondentes afirmaram não ter enfrentado dificuldades significativas com a curva de aprendizado, indicando que, ainda que o aprendizado seja autônomo, os desenvolvedores encontraram meios de assimilação das novas ferramentas.

Já em uma pergunta qualitativa a respeito do desafio da refatoração dessas novas funcionalidades, retratam *"trabalhoso"*, *"*fator humano pode introduzir erro*", "entender o que realmente pode e precisa ser mudado"* – todos esses comentários remetem ao falta ou desconhecimento por parte dos desenvolvedores de ferramentas de automação que sugerissem as mudanças necessárias diretamente no código fonte. Além disso, uma resposta particularmente elucidativa destacou:

"Sinceramente, reavaliar o desempenho (uso de recursos e latência de operações) de modo comparativo entre o antes e o depois das novas ferramentas propostas"

Ou seja, uma dificuldade técnica ao mensurar o desempenho do código ou até ausência/desinformação sobre ferramentas que avaliassem trechos de códigos. Outro participante pontuou que:

"Eu acho que dificulta um pouco a legibilidade/manutenção do código."

Essa colocação revela que a falta de conhecimento do desenvolvedor na escrita (tornando a refatoração mais complexa) ou na leitura (não saber sobre as novas features) pode afetar na legibilidade/manutenção do código para equipe. Ademais, um dos desenvolvedores Java nível profissional respondeu:

"Quando existe muito código dentro da lambda fica difícil de entender o que o código deveria fazer."

Esse comentário, retrata a necessidade de refatorar o bloco de código dentro de lambdas em funções, melhorando a coesão e legibilidade do bloco.

5.3 Impactos da Adoção: Manutenção, Produtividade e Refatoração

A percepção geral sobre os impactos positivos da adoção das novas funcionalidades foi majoritária:

- 17 pessoas reconheceram melhorias na legibilidade e manutenção do código.
- 16 relataram aumento na produtividade dos desenvolvedores.
- 12 dos 13 que participaram de refatorações avaliaram que a complexidade do código foi reduzida.

Melhoria na Legibilidade e Manutenção	Total
Sim	17
Não	5

Tabela 5.3.1 Impacto na Legibilidade e Manutenção

Impacto na Produtividade	Total
Maior produtividade	16
Nenhuma mudança	1
Menor produtividade	1

Tabela 5.3.2 Impacto na Produtividade

Participou de Refatoração?	Total
Sim	13
Não	9

Tabela 5.3.3 Participação em refatoração

Avaliação da Refatoração	Total
Reduziu a complexidade	12
Aumentou a complexidade	1

Tabela 5.3.4 Avaliação da Refatoração

Embora poucos participantes tenham relatado impactos negativos (apenas 4), houve um comentário do total de participantes que citou:

"A maior dificuldade foi garantir que tudo funcionasse em conjunto com bibliotecas que as vezes não eram compatíveis com alguns recursos. Nisso, a cobertura de testes foi fundamental para garantir que nada fosse quebrado no processo."

Esse comentário é importante por evidenciar que, mesmo entre os adeptos da modernização, há um cuidado necessário com adaptação do projeto legado com as novas funcionalidades, uma vez que pode ser mais custoso atualizar a versão do Java de um projeto em produção ao invés de atualizar o código com as novas features.

Por fim, destaca-se o depoimento de um dos respondentes sobre a recomendação da adoção para novos desenvolvedores:

"Sim, é importante aprender a tornar o código mais legível desde o princípio."

Outro participante sintetiza:

"Aprender novidades facilita muito e diminui muito do boilerplate dos códigos."

Essas percepções indicam que, embora a refatoração de sistemas legados possa exigir esforço significativo, a adoção das funcionalidades modernas do Java é amplamente vista como um investimento valioso para o futuro do projeto e da equipe.

Entretanto, boa parte dos comentários ressaltam um conhecimento sólido dos fundamentos da linguagem de programação antes de começar com o uso dessas features, uma vez que é preciso para saber o que realmente os métodos da API Streams realizam, por exemplo. Ademais foi pontuado que, sempre que houver uma refatoração em projetos legados com estas novas funcionalidades, deve-se vir acompanhada de ferramentas que assegurem o comportamento do código tais como ferramenta automatizada de testes. Outrossim, tecnologias que garantam a eficiência de desempenho dessas funcionalidades, como análises e métricas de latência e eficiência.

Já na questão sobre se as empresas deveriam adotar as novas utilidades das versões igual ou superior do Java 8, as respostas sempre envolviam a dependência de tempo, capacitação (qualificação do desenvolvedor) e custos. Exemplo de alguns trechos escolhidos:

"Não tem resposta certa para isso. A resposta varia conforme as condições da organização e dos projetos open source. Devemos lembrar que mudanças nesse tipo de escopo tem um custo operacional (tempo, mão de obra, e consequentemente, em muitos casos, dinheiro)."

Logo dependendo de prazos, consenso da equipe, treinamento e escopo, pode ser viável ou não a prioridade que as Empresas devem possuir ao refatorar códigos legados com essas novas funcionalidades.

Por outro lado, outro respondente citou a possibilidade de utilizar as funcionalidades descritas neste trabalho de acordo com a necessidade de refatoração:

"Depende. Se houver a necessidade de refatoracao por algum motivo, seja por alteração de funcionalidade ou melhoria de desempenho, então poderia reescrever utilizando as novas funcionalidades para melhorar a legibilidade do código e performance."

Portanto, ele enfatiza a possibilidade apenas quando necessário.

6. CONCLUSÃO

As funcionalidades introduzidas na linguagem Java a partir da versão 8 representam um marco significativo não apenas do ponto de vista técnico, mas também conceitual, ao promoverem uma mudança substancial na forma como os desenvolvedores interagem com a linguagem e estruturam soluções computacionais. A incorporação de recursos como expressões lambda, interfaces funcionais, Stream API, Optional e Method References proporcionou à comunidade Java ferramentas modernas, mais concisas e expressivas, aproximando a linguagem dos paradigmas da programação funcional.

A análise realizada ao longo deste trabalho evidencia que a refatoração de sistemas legados com tais recursos traz melhorias relevantes, sobretudo no que tange à legibilidade do código, à modularidade, à redução de duplicidades e à facilitação da manutenção evolutiva. Tais benefícios se estendem tanto a projetos open source quanto a sistemas corporativos, cujos casos analisados apontam para ganhos em produtividade, clareza de lógica e alinhamento com boas práticas contemporâneas de engenharia de software.

Contudo, o processo de adoção dessas funcionalidades não ocorre sem desafios. Um deles é a curva de aprendizado imposta aos desenvolvedores, especialmente àqueles mais habituados ao estilo imperativo tradicional do Java. Além disso, dificuldades técnicas relacionadas à compatibilidade com arquiteturas antigas, à integração com bibliotecas legadas e à necessidade de testes rigorosos para garantir a preservação do comportamento funcional original tornam o processo de refatoração mais criterioso e exigente.

Este trabalho também observou que a decisão de adotar os recursos do Java 8+ vai além do aspecto técnico: ela perpassa aspectos organizacionais e culturais. Em empresas, especialmente, essa transição demanda treinamentos, atualizações em pipelines de desenvolvimento e, muitas vezes, alterações em processos de qualidade e versionamento. Do lado das comunidades open source, embora a adesão a novas funcionalidades ocorra de forma mais espontânea, ainda há resistência por parte de projetos com grande acervo de código legado e baixa rotatividade de colaboradores.

Outro aspecto digno de destaque é o papel que tais recursos desempenham na renovação da própria linguagem Java, garantindo sua competitividade frente a outras linguagens modernas. Isso contribui não apenas para a longevidade do ecossistema Java, mas também para a atração de novos desenvolvedores, mais alinhados com estilos de codificação

contemporâneos.

Conclui-se, portanto, que a adoção das funcionalidades do Java 8+ deve ser incentivada, desde que conduzida com discernimento técnico e planejamento estratégico. A modernização de sistemas é um processo inevitável no ciclo de vida do software, e o Java, ao incorporar tais recursos, oferece um caminho sólido e sustentável para sua realização.

7. TRABALHOS FUTUROS E LIMITAÇÕES.

Este estudo reconhece limitações, como a não inclusão de métricas quantitativas de performance em tempo de execução, que poderiam complementar a análise qualitativa aqui realizada. Além disso, apenas foi coletado evidências de repositórios open source e empresariais e feito um estudo sobre os trechos de códigos coletados, ao invés de avaliar a refatoração das funcionalidades descritas neste trabalho nos projetos. Ademais, as perguntas do formulário foram feitas apenas sobre o código que os próprios desenvolvedores faziam, adicionando um viés e com isso tornando as avaliações preliminares e simplórias. Para pesquisas futuras, recomenda-se aprofundar os estudos sobre automação de refatorações com ferramentas baseadas em análise estática e refatoração assistida por IA, bem como explorar o impacto da adoção dessas funcionalidades em equipes de diferentes níveis de maturidade técnica. Outrossim, será feito uma análise de adoção dessas *features* em projetos open source e empresariais por meio da porcentagem de seu uso nesses projetos. Além disso, o formulário será conduzido a respeito das evidências coletadas da versão com e sem refatoração dos recursos abordados neste trabalho, a fim dos respondentes comparar as duas versões e opinar sobre a legibilidade e manutenção do trecho não refatorado em relação ao que possui as funcionalidades do escopo deste estudo.

REFERÊNCIAS

[1] VARMA, Santhosh Chitraju Gopal. The role of Java in modern software development: A comparative analysis with emerging programming languages. International Journal of Emerging Research in Engineering and Technology, v. 1, n. 2, p. 28–36, jun. 2020.

[2] ROSALES, Eduardo; BASSO, Matteo; ROSÀ, Andrea; BINDER, Walter. Profiling and Optimizing Java Streams. The Art, Science, and Engineering of Programming, v. 7, n. 3, art. 10, p. 1–39, 2023.

[3] Mazinanian, D., Tsantalis, N., & Dig, D. (2017). Understanding the Use of Lambda Expressions in Java. In Proceedings of the 33rd International Conference on Software Maintenance and Evolution (ICSME).

[4] KHATCHADOURIAN, Raffi; TANG, Yiming; BAGHERZADEH, Mehdi; RAY, Baishakhi. An Empirical Study on the Use and Misuse of Java 8 Streams. In: WEHRHEIM, Heike; CABOT, Jordi (Eds.). Fundamental Approaches to Software Engineering – FASE 2020. Lecture Notes in Computer Science, v. 12076, p. 97–118, 2020. Springer.

[5] LUCAS, Walter; FORTES, José; LOPES, Francisco Vitor; MARCÍLIO, Diego; BONIFÁCIO, Rodrigo; CANEDO, Edna Dias; LIMA, Fernanda; SARAIVA, João. Understanding the impact of introducing lambda expressions in Java programs. Journal of Software Engineering Research and Development, v. 8, n. 7, 2020.

[6] SHAPIRO, Brian. Avoiding NullPointerException in Java using Optional. Oracle, 2014. Disponível em: https://www.oracle.com/technical-resources/articles/java/java8-optional.html. Acesso em: 9 jun. 2025.

[7] PETRULIO, Fernando; SAWANT, Anand Ashok; BACCHELLI, Alberto. The indolent lambdification of Java: Understanding the support for lambda expressions in the Java ecosystem. Empirical Software Engineering, v. 26, art. 134, 2021.

[8] LUCAS, Walter; BONIFÁCIO, Rodrigo; CANEDO, Edna Dias; MARCÍLIO, Diego; LIMA, Fernanda. Does the Introduction of Lambda Expressions Improve the Comprehension of Java Programs? In: Proceedings of the XXXIII Brazilian Symposium on Software Engineering (SBES 2019), 23–27 set. 2019, Salvador, Brasil. New York: ACM, 2019. p. 1–10.

[9] Oracle. (2014). Java Platform, Standard Edition 8 API Specification. Disponível em: https://docs.oracle.com/javase/8/docs/api/ Data de acesso: 10 de abril de 2025.

[10] ROSALES, Eduardo; ROSÀ, Andrea; BASSO, Matteo; VILLAZÓN, Alex; ORELLANA, Adriana; ZENTENO, Ángel; RIVERO, Jhon; BINDER, Walter. Characterizing Java Streams in the Wild. In: Proceedings of the 26th International Conference on Engineering of Complex Computer Systems (ICECCS 2022), 2022. IEEE, p. 143–152.

[11] ORACLE. Method References. The Java™ Tutorials, Oracle. Disponível em: https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html. Acesso em: 9 jun. 2025.

[12] BAELDUNG. Java Method References. Baeldung, 15 dez. 2021. Disponível em: https://www.baeldung.com/java-method-references. Acesso em: 9 jun. 2025.

APÊNDICE - QUESTIONÁRIO

Qual é o seu nível de experiência com Java?

- ☐ Iniciante (menos de 1 ano)
- ☐ Intermediário (1 a 5 anos)
- ☐ Avançado (mais de 5 anos)

Você trabalha principalmente com:

- ☐ Desenvolvimento de software empresarial
- ☐ Desenvolvimento de software open source

Ambos

Em sua experiência profissional, qual a versão mais antiga do Java ainda em uso nos projetos em que trabalha?

- ☐ Java 6 ou anterior
- ☐ Java 7
- ☐ Java 8
- ☐ Java 11 ou superior

Seu projeto (open source ou empresarial) já adotou funcionalidades do Java 8+ (Streams, Optional, Method References, Interfaces Funcionais, Lambda Functions)?

- ☐ Sim
- ☐ Não

Caso tenha respondido "Sim" à pergunta anterior, quais dessas funcionalidades são mais utilizadas no seu ambiente de trabalho? (Marque todas as que se aplicam)

- ☐ Streams API
- ☐ Optional
- ☐ Method References
- ☐ Interfaces Funcionais
- ☐ Lambda Functions

No seu ambiente de trabalho, a adoção dessas funcionalidades foi incentivada de forma institucional (por meio de treinamentos, guias de estilo, revisões de código, etc.)?

- ☐ Sim
- ☐ Não

Você considera que a curva de aprendizado para essas novas funcionalidades foi um desafio para você ou sua equipe?

- ☐ Sim
- ☐ Não

Em sua opinião, a adoção dessas funcionalidades trouxe melhorias significativas na legibilidade e manutenção do código?

- ☐ Sim
- ☐ Não

Você percebeu algum impacto negativo na adoção dessas funcionalidades, como perda de desempenho ou dificuldades de integração com código legado?

- ☐ Sim
- ☐ Não

Em sua experiência, qual foi o maior desafio ao refatorar código legado utilizando essas novas funcionalidades? (Resposta aberta)

Você já participou da refatoração de código legado para utilizar as funcionalidades do Java 8+?

- ☐ Sim
- ☐ Não

Caso tenha participado, como você avalia o impacto da refatoração na complexidade do código?

- ☐ Reduziu a complexidade
- ☐ Permaneceu a mesma
- ☐ Aumentou a complexidade

Quanto à produtividade do desenvolvedor, a adoção das novas funcionalidades do Java 8+ trouxe:

- ☐ Maior produtividade
- ☐ Nenhuma mudança significativa
- ☐ Menor produtividade

Em sua opinião, a refatoração de código legado para adoção das novas funcionalidades do Java 8+ deve ser uma prioridade nas empresas e projetos open source? Justifique sua

resposta. (Resposta aberta)

Você recomendaria a adoção dessas funcionalidades para novos desenvolvedores que estão começando a trabalhar com Java? Por quê? (Resposta aberta)

Gostaria de compartilhar alguma experiência ou insight relevante sobre a adoção das funcionalidades do Java 8+ em projetos? (Opcional)