



SERVIÇO NACIONAL DE APRENDIZAGEM INDUSTRIAL

SENAI “GASPAR RICARDO JUNIOR”

Curso

**TÉCNICO EM DESENVOLVIMENTO
DE SISTEMAS**

**Métodos equals e hashCode em Java
e o uso de Lombok para otimizar
código em ambientes de
desenvolvimento**

Murilo Moreno Figuerôa Vieira

**Sorocaba
Novembro – 2024**



SERVIÇO NACIONAL DE APRENDIZAGEM INDUSTRIAL

SENAI “GASPAR RICARDO JUNIOR”

Murilo Moreno Figuerôa Vieira


**Métodos equals e hashCode em Java e o uso
de Lombok para otimizar código em ambientes
de desenvolvimento**

Pesquisa documentada sobre o
funcionamento dos métodos
equals e hashCode e a biblioteca
Lombok
Prof. Emerson Magalhães

Sorocaba
Novembro – 2024

SUMÁRIO

INTRODUÇÃO	4
1.1. Contextualização dos métodos equals e hashCode	4
1.2. Importância de equals e hashCode para Coleções e Frameworks como Spring 4	
1.3. Introdução ao Lombok e sua Finalidade no Desenvolvimento em Java	4
2. Fundamentos Teóricos	5
2.1. Explicação do contrato entre equals e hashCode	5
2.1.1. Regras que Governam a Implementação de equals e hashCode	5
2.1.1. Como o Contrato entre equals e hashCode Afeta o Comportamento das Coleções 6	
2.2. Importância da Implementação Correta de equals e hashCode em Entidades Java 7	
3. Utilização Prática em Coleções Java e no Spring	7
3.1. Exemplo com HashSet	7
3.2. Exemplo com HashMap	9
3.3. Exemplo Prático com Entidades Spring para Operações de Persistência e Cache 10	
3.3.1. Exemplo de Entidade Produto	10
4. Lombok: Simplificação do Código	12
4.1. Introdução à Biblioteca Lombok	12
4.2. Análise das Anotações @EqualsAndHashCode e @Data	12
4.2.1. @EqualsAndHashCode	12
4.2.2. @Data	12
4.3. Exemplo Prático de Implementação com Lombok	12
5. Vantagens e Desvantagens de Usar Lombok para equals e hashCode	13
5.1. Vantagens	13
5.1.1. Redução de código boilerplate	13
5.1.2. Legibilidade e manutenção	13
5.2. Desvantagens	13
5.2.1. Dependência de uma biblioteca externa	13
5.2.2. Dificuldade de depuração	13
CONCLUSÃO	13
BIBLIOGRAFIA	15
LISTA DE FIGURAS	16



Métodos equals e hashCode em Java e o uso de Lombok para otimizar código em ambientes de desenvolvimento

INTRODUÇÃO

1.1. Contextualização dos métodos equals e hashCode

Os métodos equals e hashCode são fundamentais no Java, pois são utilizados para definir a igualdade entre objetos e para calcular o hash que representa esses objetos em coleções como HashMap e HashSet. Em frameworks como o **Spring**, eles são essenciais para gerenciar entidades de forma única, evitando duplicações desnecessárias em cache e banco de dados.

1.2. Importância de equals e hashCode para Coleções e Frameworks como Spring

Esses métodos afetam a maneira como coleções Java armazenam e buscam objetos, e, em frameworks de persistência como Spring, garantem que entidades sejam tratadas de forma única em operações de cache e persistência. Sem implementações consistentes desses métodos, as coleções e frameworks não conseguem determinar corretamente quando dois objetos são iguais.

1.3. Introdução ao Lombok e sua Finalidade no Desenvolvimento em Java

A biblioteca **Lombok** reduz a repetição de código (boilerplate) gerando métodos comuns como equals, hashCode, toString, getters e setters. Isso torna o desenvolvimento em Java mais rápido, pois evita a necessidade de escrever manualmente esses métodos, e facilita a manutenção de projetos, especialmente em classes com muitos atributos.

2. Fundamentos Teóricos

2.1. Explicação do contrato entre equals e hashCode

2.1.1. Regras que Governam a Implementação de equals e hashCode

Quando se sobreescrevem equals e hashCode em uma classe, é necessário seguir algumas regras para garantir a coerência e evitar comportamentos inesperados em coleções:

Regra 1: Consistência de equals

- **Definição:** O método equals deve retornar consistentemente o mesmo resultado para o mesmo par de objetos durante a execução do programa, desde que esses objetos não sejam modificados.
- **Exemplo:** Se obj1.equals(obj2) é true, qualquer chamada subsequente para obj1.equals(obj2) deve continuar a retornar true enquanto os atributos que definem igualdade não mudarem.

Regra 2: Reflexividade, Simetria e Transitividade de equals

- **Reflexividade:** Um objeto deve ser igual a si mesmo, ou seja, obj1.equals(obj1) deve sempre retornar true.
- **Simetria:** Se obj1.equals(obj2) é true, então obj2.equals(obj1) também deve ser true.
- **Transitividade:** Se obj1.equals(obj2) é true e obj2.equals(obj3) é true, então obj1.equals(obj3) também deve ser true.

Regra 3: Consistência de hashCode

- **Definição:** O método hashCode deve sempre retornar o mesmo valor enquanto os atributos do objeto que definem igualdade não mudarem.
- **Exemplo:** Se dois objetos são considerados iguais pelo método equals, eles **devem** ter o mesmo hashCode.

Regra 4: Coerência entre equals e hashCode

- Se `obj1.equals(obj2)` retorna `true`, então `obj1.hashCode()` deve ser igual a `obj2.hashCode()`.
- Se `obj1.equals(obj2)` retorna `false`, não é obrigatório que `obj1.hashCode()` e `obj2.hashCode()` sejam diferentes, mas é recomendável que eles sejam distintos para reduzir colisões em coleções baseadas em hashing.

2.1.1. Como o Contrato entre equals e hashCode Afeta o Comportamento das Coleções

O contrato entre `equals` e `hashCode` é particularmente importante para o funcionamento das coleções baseadas em hashing, como `HashMap` e `HashSet`, pois essas coleções dependem desses métodos para organizar e acessar os elementos de forma eficiente.

Armazenamento e Busca:

Quando um objeto é inserido em um `HashSet`, seu `hashCode` é usado para determinar o "bucket" (espaço na memória) onde o objeto será armazenado. Caso o `hashCode` de um novo objeto seja igual ao de um objeto existente, o `HashSet` então usa o método `equals` para verificar se o novo objeto é realmente igual a algum dos já armazenados no bucket. Se `equals` retorna `true`, o `HashSet` considera os objetos como duplicados e não armazena o novo. Se `equals` retorna `false`, o novo objeto é considerado único e é armazenado.

Evitar Duplicatas:

`HashSet` usa `hashCode` e `equals` para evitar duplicatas. Sem a implementação correta de ambos os métodos, objetos duplicados podem ser armazenados, pois a coleção não conseguirá identificar corretamente objetos iguais.

2.2. Importância da Implementação Correta de equals e hashCode em Entidades Java

Quando equals e hashCode são implementados corretamente, frameworks como o Spring conseguem identificar entidades únicas e gerenciar o cache sem duplicação. Esse gerenciamento é essencial para operações de banco de dados e cache, especialmente em sistemas distribuídos, onde a consistência dos dados depende do comportamento correto desses métodos.

3. Utilização Prática em Coleções Java e no Spring

Em coleções Java baseadas em hashing, como HashSet e HashMap, os métodos equals e hashCode definem como os objetos são armazenados, localizados e comparados.

3.1. Exemplo com HashSet

O HashSet usa hashCode para localizar o "bucket" onde um objeto será armazenado, e equals para verificar se o novo objeto é realmente igual a um já existente no bucket. Isso evita duplicatas.

Imagine uma classe Pessoa com nome e idade. Vamos implementar equals e hashCode para que dois objetos Pessoa sejam considerados iguais quando têm o mesmo nome e idade.

```

1 package pctExPessoa;
2 import java.util.Objects;
3 public class Pessoa { 3 usages
4     // Atributos privados da classe 'Pessoa' que representam o nome e a idade.
5     private String nome; 4 usages
6     private int idade; 4 usages
7     // Construtor da classe 'Pessoa', inicializando os atributos 'nome' e 'idade'.
8     public Pessoa(String nome, int idade) { no usages
9         this.nome = nome;
10        this.idade = idade; }
11    /* Sobrescrevendo o método equals para definir a lógica de igualdade entre dois objetos 'Pessoa'.
12     * É fundamental para coleções como HashSet e HashMap, pois permite que as instâncias de 'Pessoa'
13     * sejam comparadas com base nos valores de 'nome' e 'idade' e não apenas nas referências de memória.
14     * @param o Objeto a ser comparado com a instância atual.
15     * @return true se os objetos tiverem o mesmo nome e idade; false caso contrário.*/
16    @Override
17    public boolean equals(Object o) {
18        // Verifica se o objeto passado é a mesma instância. Se for, retorna true imediatamente.
19        if (this == o) return true;
20        // Verifica se o objeto é uma instância de 'Pessoa'. Se não for, retorna false.
21        if (!(o instanceof Pessoa)) return false;
22        // Faz o cast seguro do objeto para 'Pessoa' para acessar seus atributos.
23        Pessoa pessoa = (Pessoa) o;
24        // Compara a idade e o nome das duas instâncias. Retorna true se ambos forem iguais.
25        return idade == pessoa.idade && nome.equals(pessoa.nome);
26    }
27    /** Sobrescrevendo o método hashCode para que ele seja consistente com o método equals.
28     * Isso é necessário para que estruturas de dados baseadas em hashing, como HashSet e HashMap,
    * funcionem corretamente. Dois objetos iguais (segundo equals) devem ter o mesmo hashCode.
    */

```

FIGURA 1 – EXEMPLO CLASSE PESSOA PARTE 1

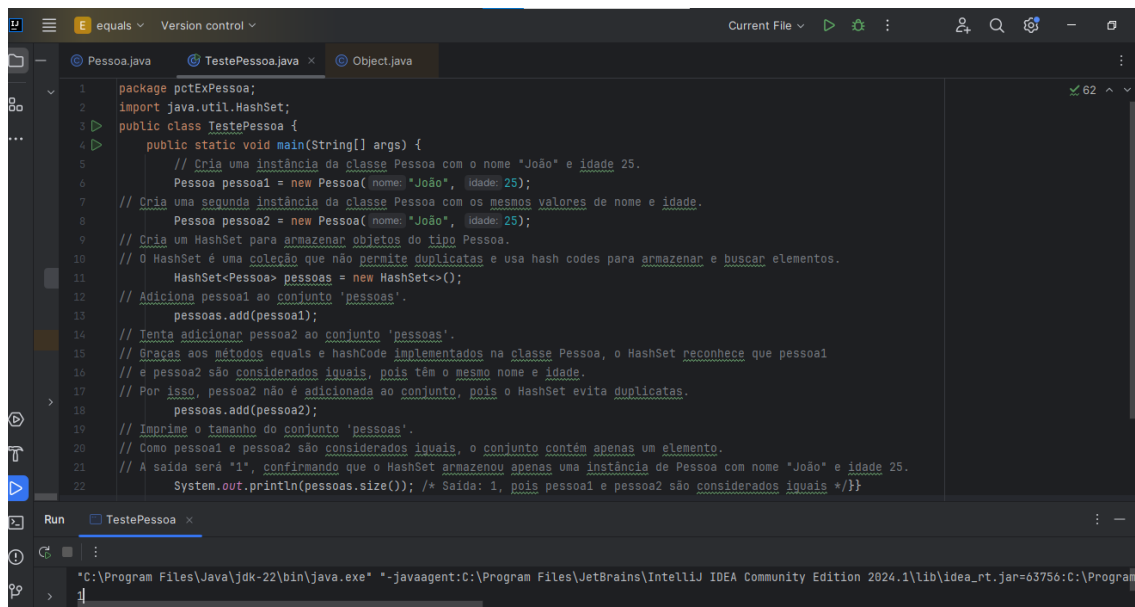
```

3 public class Pessoa { 3 usages
16 @Override
17 public boolean equals(Object o) {
18     // Verifica se o objeto passado é a mesma instância. Se for, retorna true imediatamente.
19     if (this == o) return true;
20     // Verifica se o objeto é uma instância de 'Pessoa'. Se não for, retorna false.
21     if (!(o instanceof Pessoa)) return false;
22     // Faz o cast seguro do objeto para 'Pessoa' para acessar seus atributos.
23     Pessoa pessoa = (Pessoa) o;
24     // Compara a idade e o nome das duas instâncias. Retorna true se ambos forem iguais.
25     return idade == pessoa.idade && nome.equals(pessoa.nome);
26 }
27 /** Sobrescrevendo o método hashCode para que ele seja consistente com o método equals.
28     * Isso é necessário para que estruturas de dados baseadas em hashing, como HashSet e HashMap,
29     * funcionem corretamente. Dois objetos iguais (segundo equals) devem ter o mesmo hashCode.
30     * @return o valor hash calculado com base nos atributos 'nome' e 'idade'.*/
31 @Override
32 public int hashCode() {
33     // Usa a função utilitária Objects.hash() para calcular o hashCode usando 'nome' e 'idade'.
34     // Isso garante que o hashCode seja consistente com equals.
35     return Objects.hash(nome, idade);
36 }
37 }

```

FIGURA 2 – EXEMPLO CLASSE PESSOA PARTE 2

Agora, ao adicionar duas instâncias Pessoa iguais (mesmo nome e idade) a um HashSet, apenas uma será armazenada, pois o HashSet reconhece que elas são iguais:

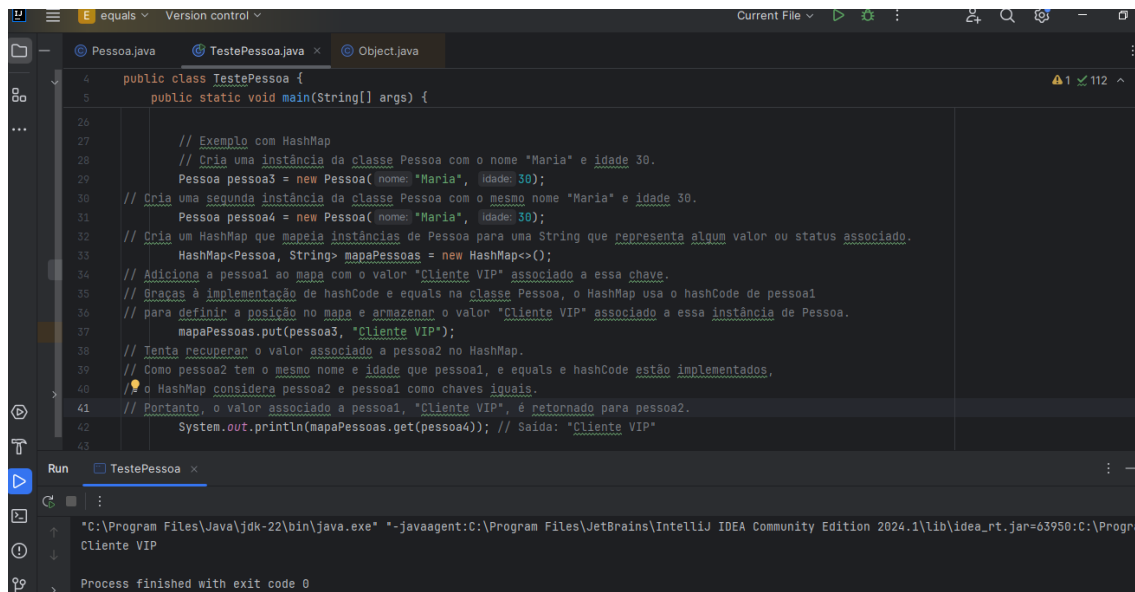


```
1 package pctExPessoa;
2 import java.util.HashSet;
3 public class TestePessoa {
4     public static void main(String[] args) {
5         // Cria uma instância da classe Pessoa com o nome "João" e idade 25.
6         Pessoa pessoa1 = new Pessoa(nome: "João", idade: 25);
7         // Cria uma segunda instância da classe Pessoa com os mesmos valores de nome e idade.
8         Pessoa pessoa2 = new Pessoa(nome: "João", idade: 25);
9         // Cria um HashSet para armazenar objetos do tipo Pessoa.
10        // O HashSet é uma coleção que não permite duplicatas e usa hash codes para armazenar e buscar elementos.
11        HashSet<Pessoa> pessoas = new HashSet<>();
12        // Adiciona pessoa1 ao conjunto 'pessoas'.
13        pessoas.add(pessoa1);
14        // Tenta adicionar pessoa2 ao conjunto 'pessoas'.
15        // Graças aos métodos equals e hashCode implementados na classe Pessoa, o HashSet reconhece que pessoa1
16        // e pessoa2 são considerados iguais, pois têm o mesmo nome e idade.
17        // Por isso, pessoa2 não é adicionada ao conjunto, pois o HashSet evita duplicatas.
18        pessoas.add(pessoa2);
19        // Imprime o tamanho do conjunto 'pessoas'.
20        // Como pessoa1 e pessoa2 são considerados iguais, o conjunto contém apenas um elemento.
21        // A saída será "1", confirmando que o HashSet armazenou apenas uma instância de Pessoa com nome "João" e idade 25.
22        System.out.println(pessoas.size()); /* Saída: 1, pois pessoa1 e pessoa2 são considerados iguais */
23    }
24 }
```

FIGURA 3 – EXEMPLO CLASSE PESSOA COM HASHSET

3.2. Exemplo com HashMap

Em um HashMap, hashCode é usado para localizar a posição onde uma chave será armazenada. equals é utilizado para comparar chaves ao buscar, garantir que valores correspondentes à mesma chave possam ser acessados.



```
1 public class TestePessoa {
2     public static void main(String[] args) {
3         // Exemplo com HashMap
4         // Cria uma instância da classe Pessoa com o nome "Maria" e idade 30.
5         Pessoa pessoa3 = new Pessoa(nome: "Maria", idade: 30);
6         // Cria uma segunda instância da classe Pessoa com o mesmo nome "Maria" e idade 30.
7         Pessoa pessoa4 = new Pessoa(nome: "Maria", idade: 30);
8         // Cria um HashMap que mapeia instâncias de Pessoa para uma String que representa algum valor ou status associado.
9         HashMap<Pessoa, String> mapaPessoas = new HashMap<>();
10        // Adiciona a pessoa1 ao mapa com o valor "Cliente VIP" associado a essa chave.
11        // Graças à implementação de hashCode e equals na classe Pessoa, o HashMap usa o hashCode de pessoa1
12        // para definir a posição no mapa e armazenar o valor "Cliente VIP" associado a essa instância de Pessoa.
13        mapaPessoas.put(pessoa3, "Cliente VIP");
14        // Tenta recuperar o valor associado a pessoa2 no HashMap.
15        // Como pessoa2 tem o mesmo nome e idade que pessoa1, e equals e hashCode estão implementados,
16        // o HashMap considera pessoa2 e pessoa1 como chaves iguais.
17        // Portanto, o valor associado a pessoa1, "Cliente VIP", é retornado para pessoa2.
18        System.out.println(mapaPessoas.get(pessoa4)); // Saída: "Cliente VIP"
19    }
20 }
```

FIGURA 4 – EXEMPLO CLASSE PESSOA COM HASHMAP

Nesse caso, pessoa4 é considerada a mesma chave que pessoa3 devido à implementação de equals e hashCode, e assim o HashMap retorna o valor correspondente.

3.3. Exemplo Prático com Entidades Spring para Operações de Persistência e Cache

No contexto do Spring, equals e hashCode são críticos para garantir que entidades sejam corretamente comparadas e manipuladas em operações de cache e persistência. Vamos examinar como isso funciona usando um exemplo de uma entidade Produto que será armazenada em cache.

3.3.1.Exemplo de Entidade Produto

Suponha que estamos criando uma entidade Produto que será armazenada em cache usando o Spring Cache, e que possui os atributos id, nome e preco. Implementamos equals e hashCode para garantir que o cache identifique corretamente quando uma instância Produto é igual a outra.

```
import javax.persistence.Entity;
import javax.persistence.Id;
import java.util.Objects;

@Entity
public class Produto {

    @Id
    private Long id;
    private String nome;
    private Double preco;

    public Produto(Long id, String nome, Double preco) {
        this.id = id;
        this.nome = nome;
        this.preco = preco;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Produto)) return false;
        Produto produto = (Produto) o;
        return id.equals(produto.id);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id);
    }

    // Getters e setters...
}
```

FIGURA 5 – EXEMPLO ENTIDADE PRODUTO SPRING

Persistência e Cache com Produto

No Spring, ao usar caches (como Redis ou EhCache) para melhorar a performance de consultas frequentes, equals e hashCode garantem que o

cache identifique corretamente os produtos como iguais ou diferentes com base no id do produto.

Imagine que temos um método de serviço que busca produtos:

```
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

@Service
public class ProdutoService {

    @Cacheable("produtos")
    public Produto buscarProdutoPorId(Long id) {
        // Simulação de uma busca no banco de dados
        return new Produto(id, "Produto Exemplo", 100.0);
    }
}
```

FIGURA 6 - SERVICE

Com o uso da anotação `@Cacheable`, o Spring armazena o produto no cache na primeira vez que é buscado. Quando o mesmo produto é solicitado novamente (ou seja, a mesma id), o Spring usa o cache para recuperar a instância, economizando recursos de banco de dados.

Teste de Cache com equals e hashCode

A implementação de `equals` e `hashCode` com base no id do produto é fundamental para o Spring identificar que produtos com o mesmo id são iguais. Sem essa implementação adequada, o cache poderia armazenar múltiplas instâncias para o mesmo Produto, o que é redundante e consome mais memória.

```
Produto produto1 = produtoService.buscarProdutoPorId(1L); // Carrega do banco e armazena
Produto produto2 = produtoService.buscarProdutoPorId(1L); // Recupera do cache

System.out.println(produto1 == produto2); // Saída: true, pois foi recuperado do cache
```

FIGURA 7 – TESTE DE CACHE

4. Lombok: Simplificação do Código

4.1. Introdução à Biblioteca Lombok

O Lombok automatiza a criação de métodos comuns, como equals, hashCode, toString, getters e setters, eliminando a necessidade de escrevê-los manualmente. Isso melhora a legibilidade do código e reduz a chance de erros, facilitando o desenvolvimento de classes modelo.

4.2. Análise das Anotações @EqualsAndHashCode e @Data

4.2.1. @EqualsAndHashCode

Gera automaticamente os métodos equals e hashCode, incluindo todos os atributos da classe por padrão, mas permitindo que campos específicos sejam incluídos ou excluídos.

4.2.2. @Data

Além de equals e hashCode, essa anotação gera toString, getters e setters para todos os campos da classe, ideal para classes simples.

4.3. Exemplo Prático de Implementação com Lombok

```
import lombok.EqualsAndHashCode;

@Data
public class Pessoa {
    private String nome;
    private int idade;

    public Pessoa(String nome, int idade) {
        this.nome = nome;
        this.idade = idade;
    }
}
```

FIGURA 8 – EXEMPLO LOMBOK

5. Vantagens e Desvantagens de Usar Lombok para equals e hashCode

5.1. Vantagens

5.1.1.Redução de código boilerplate

Lombok elimina código repetitivo, mantendo o foco nos requisitos do projeto.

5.1.2.Legibilidade e manutenção

O código fica mais limpo e focado no negócio.

5.2. Desvantagens

5.2.1.Dependência de uma biblioteca externa

Lombok é uma dependência adicional que precisa ser gerenciada.

5.2.2.Dificuldade de depuração

Como Lombok gera código em tempo de compilação, isso pode dificultar o debugging, especialmente em grandes projetos.

CONCLUSÃO

A correta implementação de equals e hashCode é fundamental para o funcionamento de coleções Java e a integridade de frameworks como o Spring. O Lombok facilita o desenvolvimento automatizando a criação desses métodos, mas seu uso em produção deve ser ponderado, especialmente considerando o impacto de depuração e a necessidade de controle sobre métodos de comparação em classes mais complexas. Para equipes familiarizadas com

Lombok, os benefícios em produtividade e organização geralmente superam as desvantagens, promovendo um código mais limpo e eficiente.

BIBLIOGRAFIA

ENTENDENDO O EQUALS E HASCODE. In: **AW.** Disponível em:<
<https://blog.algaworks.com/entendendo-o-equals-e-hashcode/> >. Acesso em: 9
nov. 2024.

EQUALS E HASCODE. In: **ANGELISKI.** Disponível em:<
<https://angeliski.com.br/equals-e-hashcode?x-host=angeliski.com.br>>. Acesso
em: 9 nov. 2024.

COMO USAR O LOMBOK EM PROJETOS JAVA. In: **DIO.** Disponível em:<
<https://www.dio.me/articles/como-usar-o-lombok-em-projetos-java>>. Acesso em:
9 nov. 2024.

PROJETO LOMBOK. In: **TREINAWEB.** Disponível em:<
[https://www.treinaweb.com.br/blog/projeto-lombok-acelerando-o-](https://www.treinaweb.com.br/blog/projeto-lombok-acelerando-o-desenvolvimento-java)
[desenvolvimento-java](https://www.treinaweb.com.br/blog/projeto-lombok-acelerando-o-desenvolvimento-java)>. Acesso em: 9 nov. 2024.

LISTA DE FIGURAS

Figura 1 – EXEMPLO CLASSE PESSOA PARTE 1 – Feito pelo autor.

Figura 2 – EXEMPLO CLASSE PESSOA PARTE 2 – Feito pelo autor.

Figura 3 – EXEMPLO CLASSE PESSOA COM HASHSET - Feito pelo autor.

Figura 4 – EXEMPLO CLASSE PESSOA COM HASHMAP – Feito pelo autor.

Figura 5 – EXEMPLO ENTIDADE PRODUTO SPRING – Feito pelo autor.

Figura 6 – SERVICE – Feito pelo autor.

Figura 7 – TESTE DE CACHE – Feito pelo autor.

Figura 8 – EXEMPLO LOMBOK – Feito pelo autor.