



**SERVIÇO NACIONAL DE APRENDIZAGEM INDUSTRIAL**

**SENAI “GASPAR RICARDO JUNIOR”**

**Curso**

**TÉCNICO EM DESENVOLVIMENTO  
DE SISTEMAS**

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

Marco Antônio da Costa Silva

Sorocaba  
Março – 2024



**SERVIÇO NACIONAL DE APRENDIZAGEM INDUSTRIAL**

**SENAI “GASPAR RICARDO JUNIOR”**

Marco Antônio da Costa Silva

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

Trabalho da matéria back-end  
sobre métodos equals e  
hashCode em java  
Prof. – Emerson Magalhães

Sorocaba  
Março – 2024

## **Introdução:**

A implementação correta dos métodos `equals` e `hashCode` em Java é crucial para o funcionamento de coleções baseadas em hashing, como `HashSet` e `HashMap`, pois permite a comparação eficiente de objetos e o acesso rápido aos dados. Frameworks como o Spring também dependem desses métodos para gerenciar entidades, garantindo a integridade dos dados em persistência e caching. A biblioteca Lombok auxilia nesse processo ao automatizar a geração de métodos como `equals` e `hashCode`, reduzindo o código repetitivo e facilitando a manutenção em projetos grandes. Este trabalho explorará o impacto do contrato entre esses métodos, seu uso no Spring, e os prós e contras de Lombok no desenvolvimento.

## Desenvolvimento:

### Fundamentos teóricos:

Em Java, o contrato entre `equals` e `hashCode` define que objetos considerados iguais pelo método `equals` devem ter o mesmo valor de `hashCode`. Isso significa que, se dois objetos são "iguais", segundo `equals`, eles precisam compartilhar o mesmo código hash para garantir consistência. Esse contrato é essencial para que as coleções que utilizam hashing funcionem corretamente.

**Reflexividade:** para qualquer referência de objeto `x`, `x.equals(x)` deve retornar `true`.

**Simetria:** para qualquer referência de objetos `x` e `y`, `x.equals(y)` deve retornar `true` se e somente se `y.equals(x)` retornar `true`.

**Transitividade:** para qualquer referência de objetos `x`, `y` e `z`, se `x.equals(y)` retorna `true` e `y.equals(z)` também retorna `true`, então `x.equals(z)` deve retornar `true`.

**Consistência:** múltiplas invocações de `x.equals(y)` devem retornar o mesmo resultado consistentemente, a menos que algum dos objetos envolvidos seja modificado.

**Consistência entre `equals` e `hashCode`:** se `x.equals(y)` retorna `true`, então `x.hashCode()` deve ser igual a `y.hashCode()`.

### Impacto nas Coleções Java:

Para uma aplicação Java, é essencial implementar `equals` e `hashCode` de forma apropriada, especialmente para classes de entidade que representam dados persistentes. Isso garante que os dados não se dupliquem e que sejam comparados adequadamente, preservando a integridade e consistência da aplicação.

## Utilização Prática em Coleções Java e no Spring

### Exemplo Prático em Coleções Java

Vamos considerar uma implementação de HashSet em que temos objetos de uma classe chamada Produto. Se Produto não possui equals e hashCode adequadamente definidos, cada nova instância de Produto será vista como um objeto único, independentemente de seus valores de atributos, o que pode resultar em elementos duplicados. Ao implementar equals e hashCode de forma correta, o HashSet pode identificar e evitar duplicatas, tornando o armazenamento mais eficiente.

### Exemplo Prático no Framework Spring

No Spring, o uso de equals e hashCode é relevante em operações de persistência, onde entidades são comparadas para assegurar a unicidade dos dados. Em sistemas de caching, equals e hashCode também são essenciais, pois permitem que o framework identifique rapidamente quais objetos já foram recuperados e devem ser mantidos em cache. Por exemplo, em um sistema de controle de estoque, onde uma classe Produto representa uma entidade de banco de dados, a implementação correta desses métodos é essencial para garantir que dois produtos com os mesmos valores de atributos sejam considerados equivalentes, assegurando a integridade dos dados.

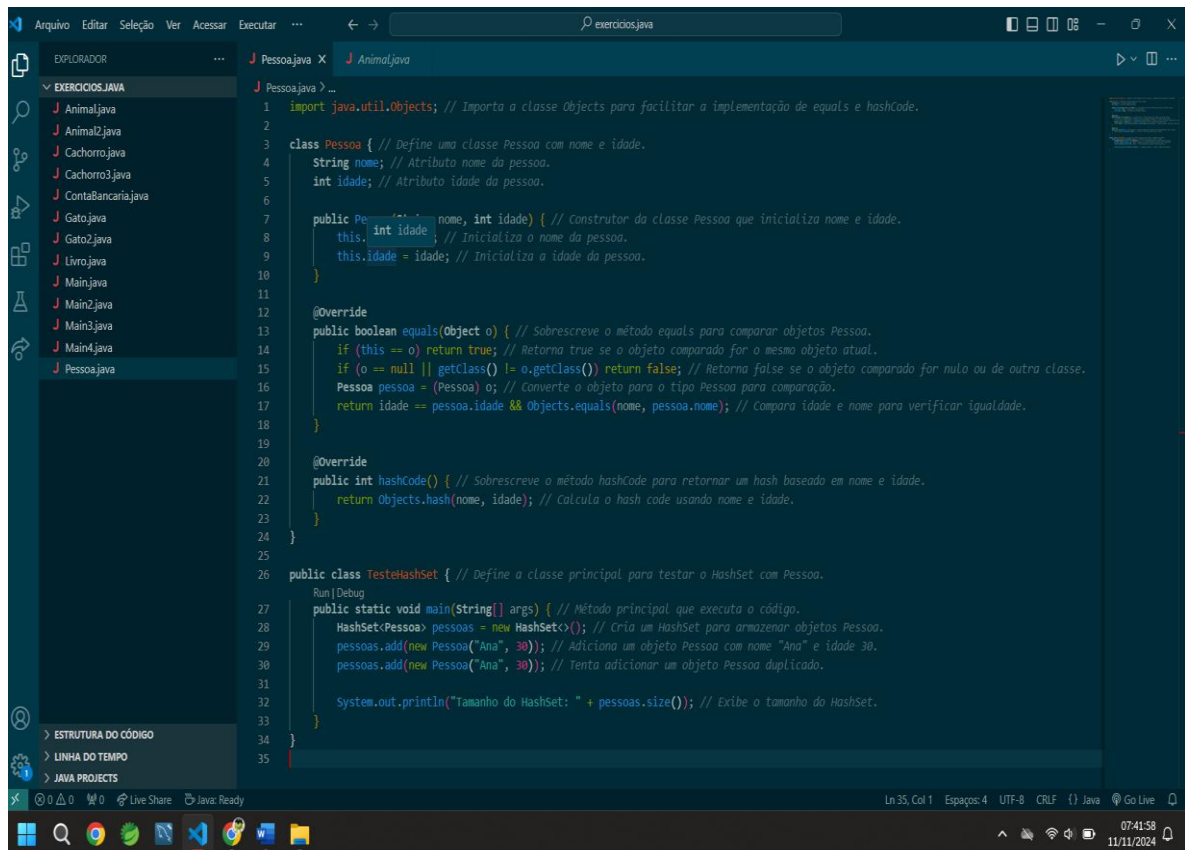
#### Exemplo 1: HashSet sem equals e hashCode:



```
1 import java.util.HashSet; // Importa a classe HashSet para o uso de uma coleção sem duplicatas.
2
3 class Pessoa { // Define uma classe Pessoa que representará pessoas com nome e idade.
4     String nome; // Atributo nome da pessoa.
5     int idade; // Atributo idade da pessoa.
6
7     public Pessoa(String nome, int idade) { // Construtor da classe Pessoa que recebe nome e idade.
8         this.nome = nome; // Inicializa o nome da pessoa.
9         this.idade = idade; // Inicializa a idade da pessoa.
10    }
11
12
13    public class TesteHashSet { // Define a classe principal para testar o HashSet com Pessoa.
14        public static void main(String[] args) { // Método principal que executa o código.
15            HashSet<Pessoa> pessoas = new HashSet<>(); // Cria um HashSet para armazenar objetos Pessoa.
16            pessoas.add(new Pessoa("Ana", 30)); // Adiciona um objeto Pessoa com nome "Ana" e idade 30.
17            pessoas.add(new Pessoa("Ana", 30)); // Tenta adicionar um objeto Pessoa duplicado.
18
19            System.out.println("Tamanho do HashSet: " + pessoas.size()); // Exibe o tamanho do HashSet.
20        }
21    }
22 }
```

**FIGURA 1:** HashSet sem equals e hashCode.

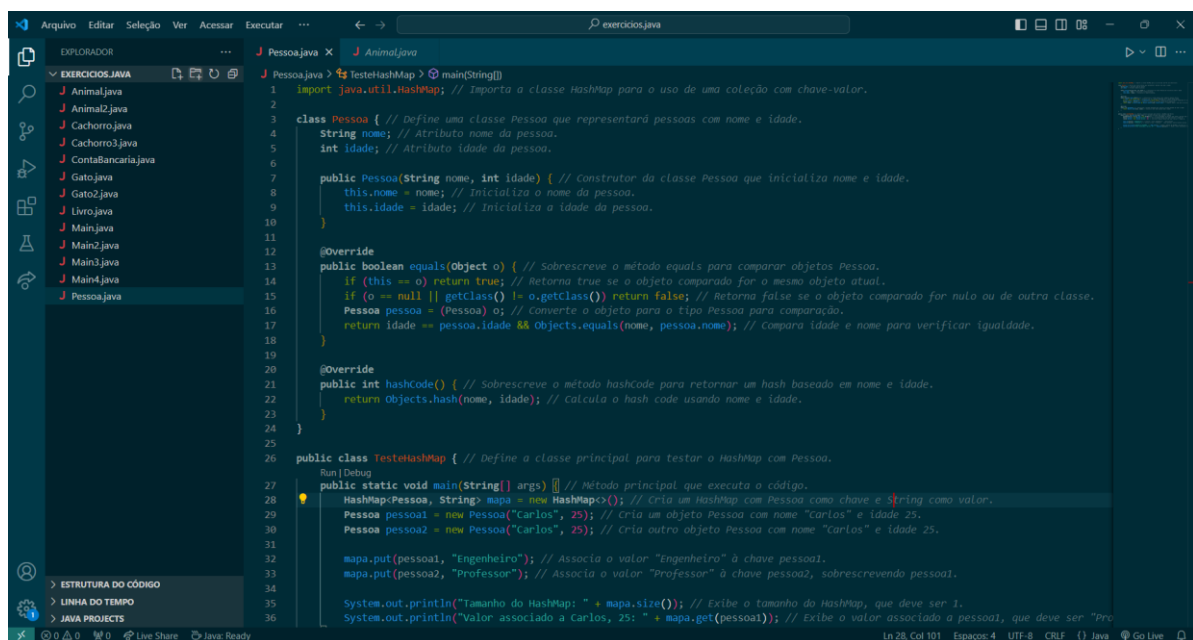
## Exemplo 2: HashSet com equals e hashCode:



```
1 import java.util.Objects; // Importa a classe Objects para facilitar a implementação de equals e hashCode.
2
3 class Pessoa { // Define uma classe Pessoa com nome e idade.
4     String nome; // Atributo nome da pessoa.
5     int idade; // Atributo idade da pessoa.
6
7     public Pessoa(String nome, int idade) { // Construtor da classe Pessoa que inicializa nome e idade.
8         this.nome = nome; // Inicializa o nome da pessoa.
9         this.idade = idade; // Inicializa a idade da pessoa.
10    }
11
12    @Override
13    public boolean equals(Object o) { // Sobrescreve o método equals para comparar objetos Pessoa.
14        if (this == o) return true; // Retorna true se o objeto comparado for o mesmo objeto atual.
15        if (o == null || getClass() != o.getClass()) return false; // Retorna false se o objeto comparado for nulo ou de outra classe.
16        Pessoa pessoa = (Pessoa) o; // Converte o objeto para o tipo Pessoa para comparação.
17        return idade == pessoa.idade && Objects.equals(nome, pessoa.nome); // Compara idade e nome para verificar igualdade.
18    }
19
20    @Override
21    public int hashCode() { // Sobrescreve o método hashCode para retornar um hash baseado em nome e idade.
22        return Objects.hash(nome, idade); // Calcula o hash code usando nome e idade.
23    }
24 }
25
26 public class TesteHashSet { // Define a classe principal para testar o HashSet com Pessoa.
27     public static void main(String[] args) { // Método principal que executa o código.
28         HashSet<Pessoa> pessoas = new HashSet<>(); // Cria um HashSet para armazenar objetos Pessoa.
29         pessoas.add(new Pessoa("Ana", 30)); // Adiciona um objeto Pessoa com nome "Ana" e idade 30.
30         pessoas.add(new Pessoa("Ana", 30)); // Tenta adicionar um objeto Pessoa duplicado.
31
32         System.out.println("Tamanho do HashSet: " + pessoas.size()); // Exibe o tamanho do HashSet.
33     }
34 }
35
```

FIGURA 2: HashSet com equals e hashCode.

## Exemplo 3: HashMap com equals e hashCode:



```
1 import java.util.HashMap; // Importa a classe HashMap para o uso de uma coleção com chave-valor.
2
3 class Pessoa { // Define uma classe Pessoa que representará pessoas com nome e idade.
4     String nome; // Atributo nome da pessoa.
5     int idade; // Atributo idade da pessoa.
6
7     public Pessoa(String nome, int idade) { // Construtor da classe Pessoa que inicializa nome e idade.
8         this.nome = nome; // Inicializa o nome da pessoa.
9         this.idade = idade; // Inicializa a idade da pessoa.
10    }
11
12    @Override
13    public boolean equals(Object o) { // Sobrescreve o método equals para comparar objetos Pessoa.
14        if (this == o) return true; // Retorna true se o objeto comparado for o mesmo objeto atual.
15        if (o == null || getClass() != o.getClass()) return false; // Retorna false se o objeto comparado for nulo ou de outra classe.
16        Pessoa pessoa = (Pessoa) o; // Converte o objeto para o tipo Pessoa para comparação.
17        return idade == pessoa.idade && Objects.equals(nome, pessoa.nome); // Compara idade e nome para verificar igualdade.
18    }
19
20    @Override
21    public int hashCode() { // Sobrescreve o método hashCode para retornar um hash baseado em nome e idade.
22        return Objects.hash(nome, idade); // Calcula o hash code usando nome e idade.
23    }
24 }
25
26 public class TesteHashMap { // Define a classe principal para testar o HashMap com Pessoa.
27     public static void main(String[] args) { // Método principal que executa o código.
28         HashMap<Pessoa, String> mapa = new HashMap<>(); // Cria um HashMap com Pessoa como chave e String como valor.
29         Pessoa pessoa1 = new Pessoa("Carlos", 25); // Cria um objeto Pessoa com nome "Carlos" e idade 25.
30         Pessoa pessoa2 = new Pessoa("Carlos", 25); // Cria outro objeto Pessoa com nome "Carlos" e idade 25.
31
32         mapa.put(pessoa1, "Engenheiro"); // Associa o valor "Engenheiro" à chave pessoa1.
33         mapa.put(pessoa2, "Professor"); // Associa o valor "Professor" à chave pessoa2, sobrescrevendo pessoa1.
34
35         System.out.println("Tamanho do HashMap: " + mapa.size()); // Exibe o tamanho do HashMap, que deve ser 1.
36         System.out.println("Valor associado a Carlos, 25: " + mapa.get(pessoa1)); // Exibe o valor associado a pessoa1, que deve ser "Pro"
37     }
38 }
39
```

FIGURA 3: HashMap com equals e hashCode.

# Lombok: Simplificação do Código

## Introdução à Biblioteca Lombok

A biblioteca Lombok é amplamente utilizada em projetos Java para reduzir código repetitivo. Com suas anotações, Lombok permite que o desenvolvedor foque na lógica do programa, sem precisar escrever manualmente métodos como equals, hashCode, toString, entre outros. Esse recurso é particularmente útil em classes de entidades que, muitas vezes, possuem grande quantidade de código repetitivo.

### Anotações @EqualsAndHashCode e @Data:

A anotação @EqualsAndHashCode gera automaticamente os métodos equals e hashCode para uma classe, seguindo o contrato padrão desses métodos. A anotação @Data, por sua vez, inclui automaticamente @EqualsAndHashCode, junto com @Getter, @Setter, @ToString, entre outras, gerando também os métodos de acesso para todos os atributos da classe.

Em uma entidade Produto, por exemplo, ao usar @Data, evitamos a necessidade de escrever manualmente os métodos equals e hashCode, simplificando o código e garantindo uma implementação consistente. Essa redução no número de linhas torna o código mais legível e diminui o risco de erros que poderiam surgir em implementações manuais.

## Vantagens e Desvantagens do Uso de Lombok para equals e hashCode:

### Vantagens:

**Redução de Código Boilerplate:** elimina a necessidade de escrever métodos repetitivos, permitindo que o desenvolvedor se concentre na lógica do programa.

**Legibilidade e Manutenção:** o código fica mais enxuto, e as atualizações se tornam mais simples, já que o Lombok cuida da implementação de métodos básicos.

**Desvantagens:**

**Dependência Externa:** ao utilizar Lombok, o projeto passa a depender de uma biblioteca externa.

**Dificuldades no Debugging:** a geração automática de código pode dificultar a depuração, especialmente em ambientes onde o código-fonte não é visível durante a execução.

**Boas Práticas para Uso do Lombok em Produção**

No uso de Lombok em ambientes de produção, é importante adotar algumas boas práticas:

**Revisão do Código Gerado:** embora Lombok automatize a geração, é essencial revisar como os métodos foram gerados.

**Uso Moderado de @Data:** para classes que exigem uma implementação mais personalizada de equals e hashCode, recomenda-se cautela com @Data, que aplica esses métodos de maneira genérica.

**Consistência de Uso:** garantir que todas as classes de um projeto sigam padrões semelhantes no uso de Lombok contribui para a coesão e facilidade de manutenção.



## **Conclusão:**

A implementação correta dos métodos `equals` e `hashCode` é essencial para o desenvolvimento de aplicações Java eficientes e para o funcionamento adequado de coleções que utilizam hashing. Em frameworks como o Spring, esses métodos desempenham um papel importante na comparação de entidades, possibilitando operações de caching e persistência de dados de forma mais segura e confiável.

O uso de Lombok oferece uma solução prática para evitar a repetição de código, melhorando a legibilidade e reduzindo o tempo de desenvolvimento. Contudo, sua utilização deve ser equilibrada, considerando-se as vantagens e desvantagens. Assim, `equals`, `hashCode` e Lombok se mostram elementos cruciais para a criação de aplicações Java eficientes, escaláveis e de fácil manutenção, sendo fundamentais para desenvolvedores que desejam otimizar o desenvolvimento sem comprometer a integridade e o desempenho do sistema.

## **Referências:**

- Documentação oficial do Java;
- Documentação oficial do Lombok;
- Artigos e materiais acadêmicos sobre implementação de coleções em Java;
- Documentação do Spring Framework sobre persistência e caching;