



**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 o código em  
ambiente de desenvolvimento**

Thafany Santos Passos

Sorocaba  
Nov – 2024



**SERVIÇO NACIONAL DE APRENDIZAGEM INDUSTRIAL**

**SENAI “GASPAR RICARDO JUNIOR”**

Thafany Santos Passos

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

Esse trabalho se trata da  
pesquisa sobre métodos os  
Equals e hashCode.  
Prof. – Emerson Magalhães

Sorocaba  
Nov – 2024

# SUMÁRIO

RESUMO .....	<b>Erro! Indicador não definido.</b>
INTRODUÇÃO.....	4
1. Método equals().....	5
1.1. SOBRESCREVENDO EQUALS().....	7
1.2. IMPLEMENTANDO UM MÉTODO EQUALS() .....	7
2. Método Hashcode .....	9
2.1. EXEMPLOS DO MÉTODO HASHCODE .....	11
2.2. CONTRATO DE HASHCODE .....	12
2.3. UTILIZAÇÃO DO HASHCODE NO SPRING FRAMEWORK .....	13
3. Coleções .....	14
3.1. HASHSET .....	14
3.2. HASHMAP.....	15
3.3. COMO O CONTRATO ENTRE EQUALS E HASHCODE AFETA O COMPORTAMENTO DAS COLEÇÕES .....	15
3.4. DEMONSTRAÇÃO DE EXEMPLOS PRÁTICOS DE EQUALS E HASHCODE APLICADOS EM COLEÇÕES COMO HASHSET E HASHMAP: .	17
Equals e hashCode em HashSet: .....	17
3.5. EXEMPLO PRÁTICO DE UMA ENTIDADE SPRING ONDE EQUALS E HASHCODE SÃO RELEVANTES PARA OPERAÇÕES DE PERSISTÊNCIA E CACHING:.....	18
4. BIBLIOTECA LOMBOK .....	19
4.1. VANTAGENS DE USAR LOMBOK: .....	19
4.2. DESVANTAGENS: .....	20
4.3. ANÁLISE DAS ANOTAÇÕES @EQUALSANDHASHCODE E @DATA... ..	21
4.4. Exemplo Prático de Implementação de uma Entidade com Lombok vs. Implementação Manual .....	22
CONCLUSÃO.....	24
BIBLIOGRAFIA .....	25
LISTA DE FIGURAS .....	<b>Erro! Indicador não definido.</b>
LISTA DE TABELAS .....	<b>Erro! Indicador não definido.</b>

# INTRODUÇÃO

Nessa pesquisa iremos abordar acerca dos métodos Equals() e hashCode além da biblioteca Lombok que possibilita otimizarmos o nosso código enquanto o desenvolvemos.

Dessa maneira, o método Equals() é um método para compararmos objetos e assim vemos se eles são iguais ou não, e o seu diferencial é que ele também consegue comparar não apenas o local de memória que os objetos ocupam mas sim seus conteúdos.

Acerca do método hashCode, ele é responsável por criar um código para o nosso objeto, e ele possui algumas coleções que permitem organizar os objetos, como o HashSet que faz uma tabela não ordenada, e o HashMap que é uma estrutura de dados que permite armazenar pares chave-valor. Cada chave é única e mapeada para um valor correspondente. Outrossim, esse método juntamente como o Equals() possibilita acessarmos e comprarmos os objetos de maneira mais eficiente e veloz.

Por fim, a respeito da biblioteca Lombok, ela é uma biblioteca em Java projetada para reduzir o código boilerplate em classes (getters, setters, equals, hashCode, e construtores).

# 1. MÉTODO EQUALS()

Em Java, comparar objetos é uma tarefa fundamental presente em muitos programas. Embora Java forneça um método `equals()` herdado da classe `Object` para comparar objetos, é essencial entender como implementar e personalizar esse método para atender às necessidades específicas do seu programa.

O método `equals()` em Java é usado para determinar se dois objetos são considerados iguais. A implementação padrão deste método na classe `Object` compara referências de objetos para ver se elas apontam para o mesmo local de memória. Porém, em muitos casos é necessário comparar o conteúdo dos objetos para determinar a igualdade. De acordo com a documentação oficial de Java, a assinatura do método `equals()` é:

```
public boolean equals(Object obj)
```

Esta assinatura indica que o método `equals()` toma um objeto como parâmetro e retorna um valor booleano indicando se os objetos são iguais ou não. Para muitas classes customizadas em Java, é necessário substituir o método `equals()` e fornecer uma implementação que compare o conteúdo dos objetos.

Vejamos um exemplo prático com a classe `Pessoa`. Suponha que queiramos comparar duas instâncias de pessoas com base em seus nomes e idades. Aqui está uma possível implementação do método `equals()` para a classe `Pessoa`:

```
public class Pessoa {  
    private String nome;  
    private int idade;  
  
    // Construtor, getters e setters  
  
    @Override  
    public boolean equals(Object obj) {  
        if (this == obj) return true;  
        if (obj == null || getClass() != obj.getClass()) return  
false;  
        Pessoa pessoa = (Pessoa) obj;  
        return idade == pessoa.idade && Objects.equals(nome,  
pessoa.nome);  
    }  
}
```

Nesta implementação, primeiro verificamos se os próprios objetos são idênticos. Em seguida, verificamos se o objeto fornecido é nulo ou não são da

mesma classe. Por fim, comparamos os campos nome e idade das duas pessoas.

Em Java, `==` e `.equals()` não são a mesma coisa. Eles têm finalidades diferentes e comportamentos distintos:

`==`: O operador `==` em Java é usado para comparar as referências de dois objetos. Ele verifica se dois objetos têm a mesma referência de memória, ou seja, se estão apontando para o mesmo local na memória. Aqui temos um exemplo utilizando esse operador:

```
String s1 = "hello";
String s2 = "hello";
String s3 = new String("hello");

System.out.println(s1 == s2); // Saída: true
System.out.println(s1 == s3); // Saída: false
```

`.equals()`: O método `.equals()` é utilizado para comparar o conteúdo de dois objetos e verificar se eles são considerados iguais de acordo com a implementação fornecida pela classe:

```
String s1 = "hello";
String s2 = "hello";
String s3 = new String("hello");

System.out.println(s1.equals(s2)); // Saída: true
System.out.println(s1.equals(s3)); // Saída: true
```

Exemplo do método `equals()`:

```
Pessoa pessoa1 = new Pessoa("Jack", 25);
Pessoa pessoa2 = new Pessoa("Jack", 25);

System.out.println(pessoa1.equals(pessoa2)); // Saída: true
```

Neste exemplo, as duas instâncias `Pessoa` possuem os mesmos valores para os campos nome e idade, portanto o método `equals()` retorna verdadeiro, o que significa que são considerados iguais.

## **SOBRESCREVENDO EQUALS()**

As classes sobrescrevem o `equals()` para garantir que dois objetos, com o mesmo conteúdo, possam ser considerados iguais.

Vale resaltar que toda comparação com `equals()` irá verificar, primeiro, se existe uma sobrescrição do mesmo em ambas as classes. Caso não haja em alguma delas, o método padrão da classe `Object` será utilizado. E assim, os objetos poderão ser considerados iguais.

Uma outra limitação que ocorrerá caso o método não seja sobrescrito, é que não será possível utilizar o objeto como chave em uma tabela hashing. Ora, se o método não foi sobrescrito, não será possível encontrar um objeto X na minha tabela. Isto é, tabela hash, queremos encontrar chaves que “batam” com o mesmo tipo de objeto que eu estou procurando. Por exemplo, foi inserido na tabela hashing uma Ferrari vermelha junto com a identificação do seu dono. Agora, preciso recuperar esta Ferrari e seu dono. E agora? Caso não se tenha mais uma referência não será possível encontrar o que desejo. Como irei mostrar um carro que sirva de exemplo para busca se tudo o que tenho está guardado na garagem? Ficaria lá, perdido para sempre.

Enfim, caso precise utilizar objetos em tabelas de hashing, será necessário sobrescrever `equals()` para que duas instâncias sejam consideradas iguais. O que poderia ser feito no exemplo anterior era sobrescrever o método `equals()` a fim de que use o CHASSI do veículo como chave para comparação. Criando um carro e setando seu chassi como X, ele irá buscar na tabela um veículo que seja chave e que tenha seu chassi como X. Isto evitará problemas com a segurança do seu projeto.

## **IMPLEMENTANDO UM MÉTODO EQUALS()**

Se sobrescrevermos o método `equals` em uma classe, ela poderia ficar desse jeito:

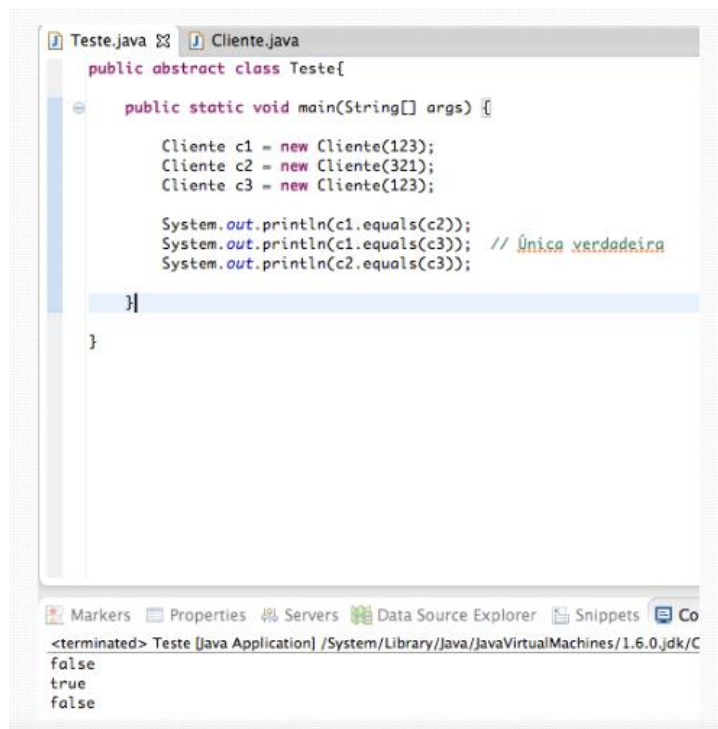


Figura 1: Saída do teste da classe com o método equals() sobrescrito

Ou seja, a classe Cliente será responsável por cadastrar novos clientes e recuperar clientes antigos de algum lugar, que poderá ser um banco de dados, uma tabela hashing ou qualquer outro mecanismo de comparação. O atributo que foi utilizado para verificar se a classe é igual à outra é o Registro Geral do cliente. A classe Teste está testando se o mecanismo realmente funciona.

Além disso, segundo a Documentação do Java, o contrato (Regras) de equals() segue as seguintes diretrizes:

- É reflexivo: para qualquer valor de referência x, x.equals() deve retornar true;
- É simétrico: para qualquer valor de referência x e y, x.equals(y) deve retornar true se, e somente se, y.equals(x) retornar true.
- É transitivo: para qualquer valor de referência de x, y e z, se x.equals(y) retornar true e y.equals(z) também retornar true, então, x.equals(z) deve retornar true.
- É consistente: para qualquer valor de referência de x e y, múltiplas chamadas de x.equals(y) retornarão consistentemente true ou consistentemente false, contanto que nenhuma informação usada nas comparações do objeto de equals tenha sido alterada.



- Para qualquer valor de referência x que não seja null, `x.equals(null)` deve retornar false.

O método `equals()` em Java é uma ferramenta poderosa para comparar objetos e determinar a igualdade. Ao compreender e implementar adequadamente esse método, você pode garantir a funcionalidade correta de seus programas Java.

## 2. MÉTODO HASHCODE

Primeiramente, precisamos entender o que é hashing e para que serve. Hashing é um processo de aplicação de uma função de hash a alguns dados. Uma função hash é apenas uma função matemática. Isso significa apenas que temos alguns dados e uma certa regra que mapeia os dados em um conjunto de caracteres (código). Por exemplo, pode ser uma cifra hexadecimal. Temos alguns dados de qualquer tamanho na entrada e aplicamos uma função hash a eles. Na saída, obtemos dados de tamanho fixo, digamos, 32 caracteres.

O resultado desse trabalho de função é chamado de código hash. Os códigos de hashing são geralmente utilizados para determinar em qual local, no conjunto, ele deve ser armazenado. Sendo, posteriormente, utilizado para fazer uma pesquisa dentro do próprio conjunto. As funções de hash são amplamente usadas em criptografia e em algumas outras áreas também. As funções hash podem ser diferentes, e elas possuem as seguintes características:

Um determinado objeto tem um determinado hashcode. Se dois objetos são iguais, seus hashcodes são os mesmos. O contrário não é verdade. Se os códigos hash forem diferentes, os objetos não são iguais com certeza.

Objetos diferentes podem ter o mesmo código hash. No entanto, é um evento muito improvável. Nesse ponto, temos uma colisão, uma situação em que podemos perder dados. A função hash "adequada" minimiza a probabilidade de colisões.

Em Java, a função hash geralmente é conectada ao método `hashCode()`. O resultado da aplicação de uma função hash a um objeto é um código hash. Assim, todo objeto Java possui um código hash. Normalmente, os

programadores substituem esse método para seus objetos, bem como relacionados a `hashCode()` o método `equals()` para processamento mais eficiente de dados específicos. O método `hashCode()` retorna um valor `int`, que é uma representação numérica do objeto.

Esse `hashCode` é usado, por exemplo, por coleções para armazenamento mais eficiente de dados e, conseqüentemente, acesso mais rápido a eles. Por padrão, o `hashCode()` função para um objeto retorna o número da célula de memória onde o objeto está armazenado. Portanto, se nenhuma alteração for feita no código do aplicativo, a função deverá retornar o mesmo valor. Se o código mudar ligeiramente, o valor `hashCode` também mudará.

Para que serve o `hashCode` em Java? Em primeiro lugar, os `hashcodes` Java ajudam os programas a serem executados mais rapidamente. Por exemplo, se compararmos dois objetos `o1` e `o2` de algum tipo, a operação `o1.equals(o2)` leva cerca de 20 vezes mais tempo do que `o1.hashCode() == o2.hashCode()`

Na classe pai `Object`, junto com o método `hashCode()`, também existe `equals()`, a função que é usada para verificar a igualdade de dois objetos. A implementação padrão dessa função simplesmente verifica os links de dois objetos quanto à sua equivalência. `equals()` e `hashCode()` têm seu contrato, portanto, se você substituir um deles, deverá substituir o outro, para não quebrar este contrato.

Dessa forma, o contrato entre `equals` e `hashCode` é crucial para o comportamento adequado das coleções baseadas em hashing, como `HashMap` e `HashSet`. Esse contrato define como as coleções identificam, armazenam e recuperam objetos, além de controlar a duplicidade. Em resumo, o contrato estabelece que:

Se dois objetos são considerados iguais pelo método `equals`, eles devem ter o mesmo `hashCode`.

Objetos que não são iguais pelo método `equals` podem, mas não são obrigados, a ter `hashCodes` diferentes.

## EXEMPLOS DO MÉTODO HASHCODE

Vamos criar uma classe `Character` com um campo — `name` . Depois disso, criamos dois objetos da classe `Character`, `character1` e `character2` e os definimos com o mesmo nome. Se usarmos o padrão `hashCode()` e `equals()` da classe `Object` , definitivamente obteremos objetos diferentes, não iguais. É assim que funciona o `hashCode` em Java. Eles terão `hashcodes` diferentes porque estão em células de memória diferentes e o resultado da operação `equals()` será falso.

```
1  import java.util.Objects;
2
3  public class Character {
4      private String Name;
5
6      public Character(String name) {
7          Name = name;
8      }
9
10     public String getName() {
11         return Name;
12     }
13
14     public void setName(String name) {
15         Name = name;
16     }
17
18     public static void main(String[] args) {
19         Character character1 = new Character("Arnold");
20         System.out.println(character1.getName());
21         System.out.println(character1.hashCode());
22         Character character2 = new Character("Arnold");
23         System.out.println(character2.getName());
24         System.out.println(character2.hashCode());
25         System.out.println(character2.equals(character1));
26     }
27 }
```

E se quisermos ter objetos iguais se eles tiverem os mesmos nomes? O que deveríamos fazer? A resposta: devemos sobrescrever os métodos `hashCode()` e `equals()` da classe `Object` para nossa classe `Character`.

No caso do nosso exemplo temos o seguinte código:

```

1  import java.util.Objects;
2
3  public class Character {
4      private String Name;
5
6      public Character(String name) {
7          Name = name;
8      }
9
10     public String getName() {
11         return Name;
12     }
13
14     public void setName(String name) {
15         Name = name;
16     }
17
18     @Override
19     public boolean equals(Object o) {
20         if (this == o) return true;
21         if (!(o instanceof Character)) return false;
22
23         Character character = (Character) o;
24
25         return getName() != null ? getName().equals(character.getName()) : character.getName()
26     }
27
28     @Override
29     public int hashCode() {
30         return getName() != null ? getName().hashCode() : 0;
31     }
32
33     public static void main(String[] args) {
34         Character character1 = new Character("Arnold");
35         System.out.println(character1.getName());
36         System.out.println(character1.hashCode());
37         Character character2 = new Character("Arnold");
38         System.out.println(character2.getName());
39         System.out.println(character2.hashCode());
40         System.out.println(character2.equals(character1));
41     }
42 }

```

## CONTRATO DE HASHCODE

De acordo com a documentação do Java, existe um contrato a ser seguido caso seja sobrescrito o hashCode():

Sempre que for chamado no mesmo objeto mais de uma vez durante a execução de um aplicativo Java, o método hashCode() terá que retornar consistentemente o mesmo inteiro, contanto que nenhuma informação usada nas comparações de equals() envolvendo o objeto tenha sido alterada. Este

inteiro terá que permanecer constante de uma execução a outra do mesmo aplicativo.

Se dois objetos forem iguais de acordo com o método `equals(Object)`, então, a chamada do método `hashCode()` nos dois objetos deve produzir como resultado o mesmo inteiro.

Não é obrigatório que quando dois objetos forem diferentes de acordo com o método `equals(Object)`, a chamada ao método `hashCode()` nesses objetos produza resultados inteiros distintos. No entanto, o programador deve ficar alerta para o fato de que produzir resultados inteiros distintos para objetos diferentes pode melhorar o desempenho das tabelas de hashing.

## **UTILIZAÇÃO DO HASHCODE NO SPRING FRAMEWORK**

No Spring Framework, `hashCode()` e `equals()` desempenham papéis importantes, especialmente quando se trabalha com coleções e beans gerenciados pelo container de inversão de controle (IoC).

### **1. Beans Únicos e Controle de Dependências:**

O Spring `hashCode()` e `equals()` para garantir que algumas coisas únicas sejam identificadas corretamente. Se o container de Spring gerencia múltiplas instâncias, ele pode usar o `hashCode()` para garantir que cada uma seja única ou para verificar a presença de algo específico em suas coleções internas.

### **2. Anotações como @Component e @Repository:**

As anotações como `@Component`, `@Repository`, `@Service`, ou `@Controller` frequentemente dependem de `hashCode()` e `equals()` para garantir que instâncias idênticas não sejam duplicadas ao serem injetadas em diferentes partes do aplicativo.

### **3. Implementação em Cache com Spring Cache:**

O Spring Cache usa `hashCode()` para armazenar e recuperar dados em cache eficientemente. O código hash ajuda a localizar a entrada de cache com base na chave, permitindo que os dados sejam recuperados mais rapidamente.

### 3. COLEÇÕES

Desde as primeiras versões, Java dispõe das estruturas de arrays e as classes Vector e Hashtable. No entanto, além da dificuldade em implementar estruturas de dados utilizando arrays, os desenvolvedores sentiam falta de classes que implementassem estruturas como listas ligadas e tabelas de espalhamento (hash). Para atender a essas necessidades, a partir de Java 1.2, foi criado um conjunto de interfaces e classes denominado Collections Framework, que faz parte do pacote java.util.

Vale ressaltar que, A Hashtable é uma das primeiras estruturas de dados fornecidas pelo Java, existente desde a sua primeira versão. A sua capacidade de armazenar pares chave-valor a torna indispensável para muitas aplicações que necessitam de armazenamento e recuperação eficientes de dados. Ela é sincronizada, não permite valores nulos e possui muita eficiência.

Dessa maneira, Collections Framework é um conjunto bem definido de interfaces e classes para representar e tratar grupos de dados como uma única unidade, que pode ser chamada coleção, ou collection.

Acerca do hash, há as chamadas tabelas de espalhamento que utiliza das implementações HashSet e HashMap.

#### **HASHSET**

A classe HashSet implementa a interface Set e é baseada em uma tabela hash. Com o HashSet os elementos não estão ordenados, não há garantia de que os elementos estarão na mesma ordem após algum tempo e as operações de adição, exclusão e busca serão realizadas em tempo constante.

Alguns pontos importantes sobre HashSet:

- Porque a classe implementa a interface Set, ela só pode armazenar valores únicos;
- Pode armazenar valores NULL;
- A ordem em que os elementos são adicionados é calculada usando um código hash;

Exemplo de uma saída da implementação do HashSet:

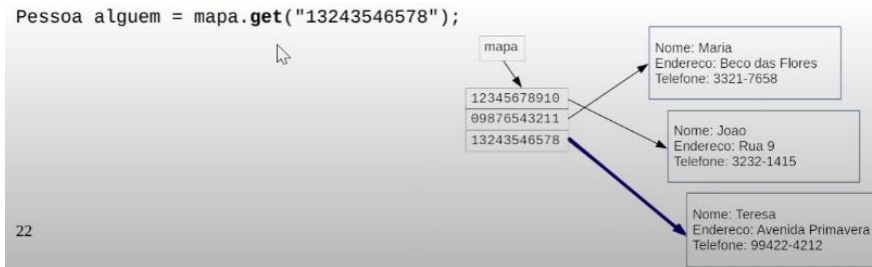
```
Manga{id=2, nome='Dragon ball Z', preco=2.99, quantidade=0}  
Manga{id=3, nome='Attack on titan', preco=11.2, quantidade=2}  
Manga{id=1, nome='Berserk', preco=9.5, quantidade=5}  
Manga{id=5, nome='Hellsing Ultimate', preco=19.9, quantidade=0}  
Manga{id=4, nome='Pokemon', preco=3.2, quantidade=0}
```

## HASHMAP

É uma estrutura de dados que permite armazenar pares chave-valor. Cada chave é única e mapeada para um valor correspondente. Isso permite que você associe dados (valores) a identificadores (chaves) de maneira eficiente, permitindo recuperação rápida dos valores com base nas chaves.

Exemplo da implementação do HashMap:

```
HashMap<String, Pessoa> mapa = new HashMap<String, Pessoa>();  
mapa.put("12345678910", joao);  
mapa.put("09876543211", maria);  
mapa.put("13243546578", teresa);  
Pessoa alguem = mapa.get("13243546578");
```



## COMO O CONTRATO ENTRE EQUALS E HASHCODE AFETA O COMPORTAMENTO DAS COLEÇÕES

Esse contrato influencia o comportamento das coleções de várias maneiras:

### 1. HashSet e Controle de Duplicidade

No caso do HashSet, a coleção usa hashCode para determinar a posição onde um objeto será armazenado e equals para verificar se ele já existe.

- Ao adicionar um objeto: O HashSet calcula o hashCode do objeto para localizar uma posição onde ele poderia ser armazenado. Em seguida, usa equals para verificar se já existe um objeto equivalente nessa posição.
- Se hashCode e equals estão implementados corretamente: Objetos iguais serão armazenados apenas uma vez, mantendo a unicidade.

- Se o contrato não for seguido: Se equals considera dois objetos iguais, mas hashCode retorna valores diferentes, os objetos podem ser armazenados em buckets distintos. Isso resulta em duplicidade indesejada no conjunto.

## **2. HashMap e Mapeamento de Chaves**

Para HashMap, o contrato entre equals e hashCode afeta como as chaves são armazenadas e recuperadas.

- Ao inserir uma chave-valor: O HashMap usa o hashCode da chave para localizar a posição onde o par chave-valor deve ser armazenado. Dentro disso, ele usa equals para identificar se uma chave igual já existe.
- Implementação correta: Se duas chaves são iguais (de acordo com equals), elas devem ter o mesmo hashCode, e o HashMap substituirá o valor anterior pela nova entrada.
- Violação do contrato: Se hashCode não é consistente com equals, chaves consideradas iguais por equals podem ter hashCodes diferentes e acabar em buckets diferentes. Isso faz com que HashMap trate as chaves como distintas, levando a armazenamento incorreto e dificuldades na recuperação de valores.

## **3. Eficiência e Colisões**

Coleções de hashing dependem de uma distribuição uniforme dos hashCodes para reduzir colisões, onde múltiplos objetos caem na mesma posição. Colisões afetam o desempenho, pois a coleção precisará usar equals para comparar todos os objetos na mesma posição.

Hash code com boa distribuição: Gera menos colisões, tornando a busca e a inserção mais eficientes.

Muitos objetos com o mesmo hash code: Isso causa colisões frequentes, o que degrada o desempenho das operações.

Nesse interím, sempre é preciso substituir também o método hashCode() ao implementar equals() para garantir a consistência de estruturas de dados baseadas em hash, como HashMap e HashSet.



## DEMONSTRAÇÃO DE EXEMPLOS PRÁTICOS DE EQUALS E HASHCODE APLICADOS EM COLEÇÕES COMO HASHSET E HASHMAP:

Equals e hashCode em HashSet:

```
import java.util.HashSet;
import java.util.Objects;

public class Produto {
    private int id;
    private String nome;

    public Produto(int id, String nome) {
        this.id = id;
        this.nome = nome;
    }

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

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

    @Override
    public String toString() {
        return "Produto{" + "id=" + id + ", nome='" + nome + '\'' + '}';
    }
}
```

```
public static void main(String[] args) {
    HashSet<Produto> produtos = new HashSet<>();

    produtos.add(new Produto(1, "Teclado"));
    produtos.add(new Produto(2, "Mouse"));
    produtos.add(new Produto(1, "Teclado"));

    System.out.println(produtos);
}
}
```

Neste exemplo temos que:

- Dois produtos com o mesmo id são considerados iguais.

- Ao adicionar produtos ao HashSet, o segundo produto com id = 1 não será adicionado, porque o HashSet detecta que ele é igual ao produto já existente, garantindo a unicidade.

## EXEMPLO PRÁTICO DE UMA ENTIDADE SPRING ONDE EQUALS E HASHCODE SÃO RELEVANTES PARA OPERAÇÕES DE PERSISTÊNCIA E CACHING:

Em uma aplicação Spring, vamos definir uma entidade Cliente com id e email como propriedades únicas. A implementação de equals e hashCode é importante para garantir a consistência no uso dessa entidade em operações de caching e coleções.

Entidade Cliente com equals e hashCode:

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

@Entity
public class Cliente {
    @Id
    private Long id;
    private String nome;
    private String email;

    // Getters e setters

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Cliente cliente = (Cliente) o;
        return Objects.equals(id, cliente.id) && Objects.equals(email, cliente.email);
    }

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

Uso de Cliente em Operações de Persistência e Caching:

Imagine que temos um serviço que utiliza cache para reduzir consultas repetidas ao banco de dados. Ao buscar ou atualizar um cliente pelo ID, os

métodos equals e hashCode ajudam a evitar inconsistências e garantir o cache correto.

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

@Service
public class ClienteService {

    @Cacheable("clientes")
    public Cliente buscarClientePorId(Long id) {
        // Simulação de uma busca ao banco de dados
        return clienteRepository.findById(id).orElseThrow(() -> new RuntimeException("Cliente não encontrado"));
    }
}
```

Neste exemplo percebemos que:

- Quando o método buscarClientePorId é chamado pela primeira vez, o cliente é armazenado no cache.
- Em chamadas subsequentes com o mesmo id, o cache será usado, evitando novas consultas ao banco.
- Os métodos equals e hashCode asseguram que o cache retorne sempre o objeto correto, pois eles garantem que Cliente seja tratado de forma única por seu id e email.

## 4. BIBLIOTECA LOMBOK

Lombok é uma biblioteca Java projetada para reduzir o código boilerplate em classes, como getters, setters, equals, hashCode, e construtores. Lombok usa anotações para gerar automaticamente esses métodos, mantendo o código mais limpo e reduzindo erros comuns.

### VANTAGENS DE USAR LOMBOK:

- Redução de Código Boilerplate: Lombok elimina a necessidade de escrever métodos repetitivos e padrões.
- Legibilidade: Com menos código boilerplate, as classes se tornam mais legíveis, destacando apenas o que é realmente importante.

- **Manutenção Simples:** Lombok reduz o risco de bugs ao centralizar a geração de métodos, tornando a manutenção mais simples e segura.
- **Aumento de Produtividade:** A codificação se torna mais rápida, pois Lombok cria automaticamente métodos comuns, permitindo que os desenvolvedores foquem na lógica de negócios.

## **DESVANTAGENS:**

- **Dependência Externa:** O projeto depende do Lombok, e mudanças na biblioteca ou na compatibilidade podem impactar o código. Em ambientes de produção, qualquer atualização ou bug na biblioteca pode impactar a estabilidade da aplicação.
- **Dificuldade em Depurar:** Como o código é gerado em tempo de compilação, depurar `equals` e `hashCode` gerados pode ser mais complexo, especialmente em IDEs que não suportam bem o Lombok.
- **Problemas com Customização:** Embora seja possível personalizar, o uso de anotações pode ser limitado em relação à flexibilidade de um código manual, especialmente em casos complexos de comparação entre objetos.


Além disso o Lombok apresenta algumas desvantagens e questões de boas práticas como as apresentadas a seguir:

- **Ocultação de Código:** Como Lombok gera código automaticamente, desenvolvedores podem perder visibilidade dos métodos gerados. Isso dificulta a depuração e compreensão completa do comportamento do código, especialmente em métodos que desempenham papel crucial em coleções (como `HashMap` e `HashSet`), onde uma implementação inadequada de `equals` e `hashCode` pode causar problemas sérios de lógica.
- **Problemas de Manutenção a Longo Prazo:** Se o código precisar ser mantido por uma equipe que não está familiarizada com Lombok ou se a biblioteca não estiver documentada na empresa, a curva de aprendizado pode se tornar uma barreira. Além disso, em versões futuras do Java, atualizações ou incompatibilidades de Lombok podem exigir refatorações significativas.

## ANÁLISE DAS ANOTAÇÕES @EQUALSANDHASHCODE E @DATA

A anotação @EqualsAndHashCode do Lombok gera automaticamente os métodos equals e hashCode. Ela é altamente personalizável e permite especificar quais campos serão considerados para a comparação, se necessário.

Exemplo de @EqualsAndHashCode:




```
import lombok.EqualsAndHashCode;

@EqualsAndHashCode
public class Produto {
    private int id;
    private String nome;
}
```

- Aqui, equals e hashCode serão gerados considerando todos os campos da classe.

Podemos customizar para que apenas o campo id seja usado:



```
@EqualsAndHashCode(of = "id")
public class Produto {
    private int id;
    private String nome;
}
```

A anotação @Data é uma combinação de várias anotações do Lombok, gerando automaticamente getters, setters, equals, hashCode, e toString, além de um construtor para todos os campos finais (final).

Exemplo de @Data:

```
import lombok.Data;

@Data
public class Cliente {
    private int id;
    private String nome;
}
```

Aqui, todos os métodos básicos (getters, setters, equals, hashCode, toString) serão gerados para a classe Cliente, simplificando ainda mais o código.

## Exemplo Prático de Implementação de uma Entidade com Lombok vs. Implementação Manual

Implementação com Lombok:

```
import lombok.Data;

@Data
public class Produto {
    private int id;
    private String nome;
}
```

Com @Data, evitamos a necessidade de escrever manualmente todos os métodos. O código é simplificado para apenas a definição dos campos.

Implementação manual:

```
import java.util.Objects;

public class Produto {
    private int id;
    private String nome;

    public Produto(int id, String nome) {
        this.id = id;
        this.nome = nome;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Produto produto = (Produto) o;
        return id == produto.id && Objects.equals(nome, produto.nome);
    }

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

## CONCLUSÃO

Portanto, nessa pesquisa exploramos mais desses conceitos e suas aplicações no Spring, afim de desenvolvermos os códigos de maneira cada vez mais eficiente e produtiva.

Isto é, vimos que o método `Equals()` é uma maneira muito eficiente de compararmos conteúdos de um objeto e reduzimos a chances de duplicações e outros problemas. Também observamos que, o método `HashCode` permite uma identificação dos objetos, uma vez que gera códigos para cada um deles, e assim permite maior agilidade para acessarmos esses objetos. Esses dois métodos juntos permitem compararmos e acessarmos todos os objetos de maneira objetiva e assertiva. Ademais, existem as coleções que possibilitam organizar os objetos de acordo com o `HashCode`, como o `HashSet` e `HashMap`. Ou seja, sem esses métodos poderá haver muitos erros no que se refere a comparação, identificação e acesso dos objetos.

Além disso, nota-se que a biblioteca Lombok permite reduzir os códigos boilerplate, como os `Equals` e `HashCode`, com o uso de anotações (`@EqualsAndHashCode` e `@Data`) e que isso possibilita o aumento da produtividade, porém também foi possível ver que ele possui algumas desvantagens, principalmente em relação ao ambiente de produção, como o problema de customização, a dependência de uma biblioteca externa, a ocultação do código e o problema com a manutenção a longo prazo.



## BIBLIOGRAFIA

DE ARGOLO AZEVEDO, T. **Sobrescrevendo o método hashCode() em Java**. Disponível em: <<https://www.devmedia.com.br/sobrescrevendo-o-metodo-hashcode-em-java/26488>>. Acesso em: 9 nov. 2024b.

SANTOS, A. **Equals() em Java: Compreendendo e Implementando a Comparação de Objetos**. Disponível em: <<https://www.dio.me/articles/equals-em-java-compreendendo-e-implementando-a-comparacao-de-objetos>>. Acesso em: 9 nov. 2024.

DE ARGOLO AZEVEDO, T. **Sobrescrevendo o método equals() em Java**. Disponível em: <<https://www.devmedia.com.br/sobrescrevendo-o-metodo-equals-em-java/26484>>. Acesso em: 9 nov. 2024.

ARAUJO, C. **Java Collections: Como utilizar Collections**. Disponível em: <<https://www.devmedia.com.br/java-collections-como-utilizar-collections/18450>>. Acesso em: 09 nov. 2024.

JAVARUSH **HashSet em Java**. Disponível em <<https://javarush.com/pt/groups/posts/pt.2147.hashset-em-java>>. Acesso em: 09 nov. 2024.

NETTO, H. V. **Coleções em java: ArrayList, HashSet e HashMap**. Disponível em: <[https://www.youtube.com/watch?v=UMuUlqK7i\\_Y](https://www.youtube.com/watch?v=UMuUlqK7i_Y)>. Acesso em: 10 nov. 2024.

SILVA, N. **Classe Hashtable em Java. Caffeine Algorithm**, 21 Sep. 2023. Disponível em: <<https://caffeinealgorithm.com/blog/classe-hashtable-em-java>>. Acesso em: 09 nov. 2024.

DEVDOJO. **175 - Coleções pt 15 - Set, HashSet**. Disponível em: <<https://www.youtube.com/watch?v=dJ3fhRq-5tM>>. Acesso em: 09 nov. 2024.

SERRA, A. **Vou explicar o HashMap em Java de forma simples, pode ser?** Disponível em: <<https://dev.to/antoniorws/vou-explicar-o-hashmap-em-java-de-forma-simples-pode-ser-2c9g>>. Acesso em: 09 nov. 2024.

BOMFIM, F. **Como Lombok Pode Transformar Seu Código Java**. Disponível em: <<https://www.dio.me/articles/como-lombok-pode-transformar-seu-codigo-java>>. Acesso em: 09 nov. 2024.

JAVARUSH **Biblioteca Lombok**. Disponível em: <<https://javarush.com/pt/groups/posts/pt.2753.biblioteca-lombok>>. Acesso em: 09 nov. 2024.

JAVARUSH **Java hashCode()** Disponível em: <<https://codegym.cc/pt/groups/posts/pt.210.java-hashcode->>. Acesso em: 09 nov. 2024.