Universidade Federal de Ouro Preto

CSI032 - Programação de Computadores II

Collections: Set

Professor: Dr. Rafael Frederico Alexandre
Contato: rfalexandre@decea.ufop.br
Colaboradores: Eduardo Matias Rodrigues
Contato: eduardo.matias@aluno.ufop.edu.br

Collections framework

Collections

Set e Map
introdução

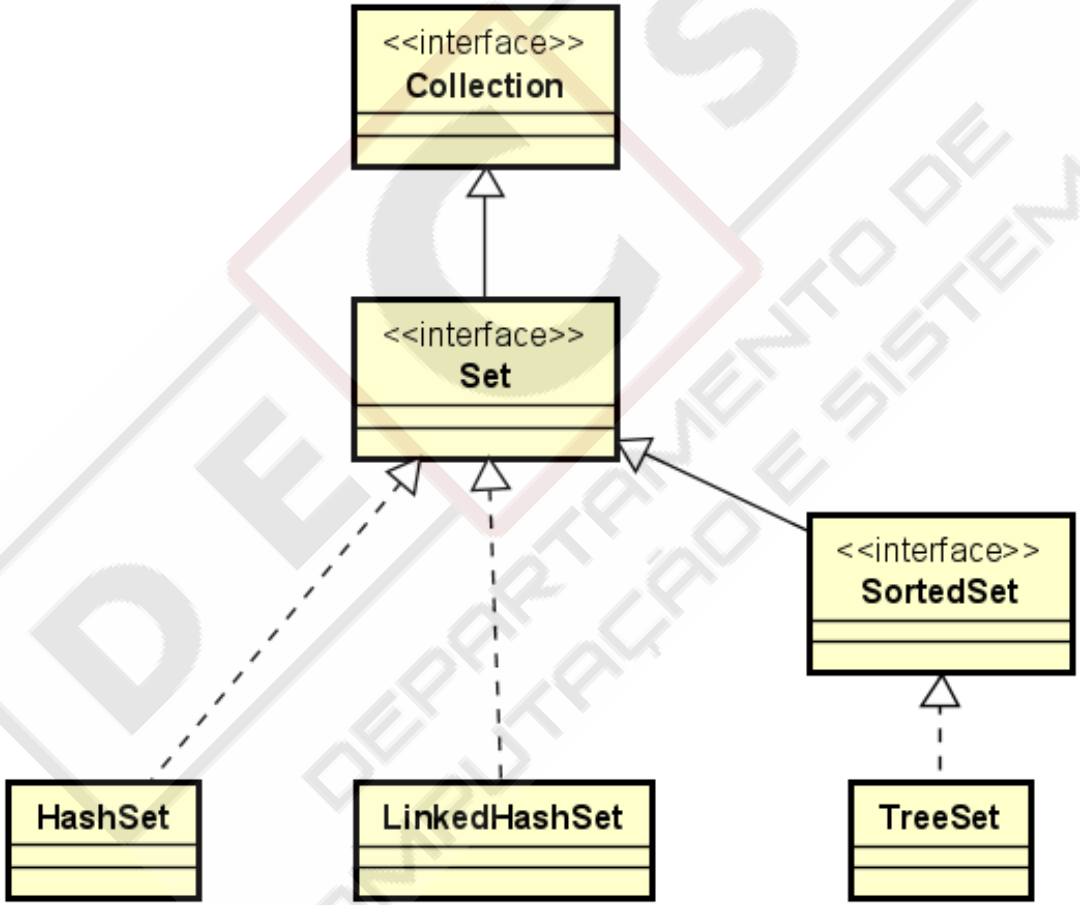
1. Trabalharemos agora com a Interface **Set** presente no framework Collections;
2. Veremos como implementar essa estrutura e as vantagens e desvantagens de utilizá-la;

Collections

Set
Conjuntos

1. A interface **Set** representa um conjunto como o qual conhecemos na matemática:
   1. coleção não ordenada de elementos;
   2. não admite elementos duplicados.

Set
Diagrama de classes



Set
HashSet e TreeSet

1. **HashSet** armazena seu elementos em uma *tabela de hash*, não há garantias que seus dados tenha alguma ordem, escolha um HashSet se você precisar de elementos únicos onde a ordem não importa;
2. **TreeSet** nesse conjunto a ordem dos elementos é classificada, por ordem natural (se for um conjunto de números eles estão em ordem crescente) ou em uma ordem arbitrátia utilizando um comparador;
    1. utiliza uma árvore para armazenar os dados;
    2. acesso é mais lento comparado ao **HashSet**.

Set
LinkedHashSet

1. **LinkedHashSet** Mantém a ordem de inserção de seus elementos;
    1. Utiliza uma lista duplamente encadeada para armazenar os dados.

Set
HashSet

1. Iremos implementar apenas a HashSet;
2. Vide a documentação para mais detalhes sobre LinkedHashSet e TreeSet;
3. LinkedHashSet:

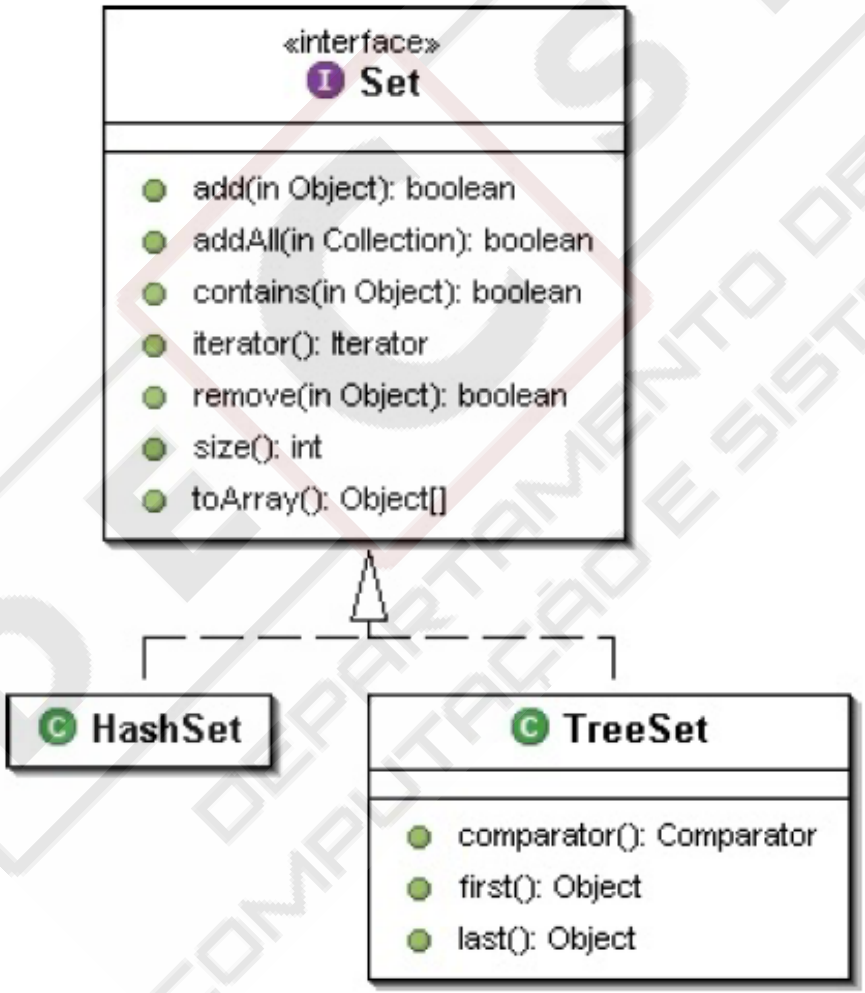
`https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html`

1. TreeSet:

`https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html`

Set

Diagrama de classes: [Caelum, 2017]



Set
Operações

1. Possíveis operações para se fazer sobre um conjunto:
    1. Adicionar um novo elemento ao conjunto;
    2. Verificar se um determinado elemento pertence ao conjunto;
    3. Remover um elemento;
    4. Limpar conjunto;

Adicionar elemento ao conjunto

add

```java
public static void main(String[] args) {

    Set<String> cidades = new HashSet<>();

    cidades.add("João Monlevade");
    cidades.add("Coronel Fabriciano");
    cidades.add("Belo Horizonte");
    cidades.add("João Monlevade"); // Repetido, não vai inserir

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

```
[João Monlevade, Coronel Fabriciano, Belo Horizonte]
```

Adicionar elemento ao conjunto
add

1. Método *add* retorna um *boolean* indicando se o elemento foi inserido no conjunto.

```
public static void main(String[] args) {

    Set<String> cidades = new HashSet<>();
    String[] nomesCidades = {"João Monlevade", "Coronel Fabriciano",
                             "Belo Horizonte", "Ouro preto", "Viçosa",
                             "João Monlevade"};

    for (String cidade : nomesCidades) {
        if (!cidades.add(cidade)) {
            System.out.println(cidade + " já esta no conjunto, "
                               + "não será inserida novamente");
        }
    }
    System.out.println("\nCidades");
    System.out.println(cidades);
}
```

Saída do programa
add

```
João Monlevade já esta no conjunto, não será inserida novamente

Cidades
[João Monlevade, Coronel Fabriciano, Ouro preto, Viçosa, Belo Horizonte]
```

Remover elemento

remove

```
public static void main(String[] args) {

    Set<String> cidades = new HashSet<>();
    String[] nomesCidades = {"João Monlevade", "Coronel Fabriciano",
                             "Belo Horizonte", "Ouro preto", "Viçosa",
                             "João Monlevade"};

    for (String cidade : nomesCidades) {
        cidades.add(cidade);
    }

    System.out.println(cidades);
    cidades.remove("Viçosa");
    System.out.println(cidades);
}
```

Saída do programa

remove

```
[João Monlevade, Coronel Fabriciano, Ouro preto, Viçosa, Belo Horizonte]
[João Monlevade, Coronel Fabriciano, Ouro preto, Belo Horizonte]
```

Remover elemento

remove

1. Assim como o método *add* o *remove* retorna um boolean indicando o sucesso da operação;

```
public static void main(String[] args) {

    Set<String> cidades = new HashSet<>();
    String[] nomesCidades = {"João Monlevade", "Coronel Fabriciano",
                             "Belo Horizonte", "Ouro preto", "Viçosa",
                             "João Monlevade"};

    for (String cidade : nomesCidades) {
        cidades.add(cidade);
    }

    System.out.println(cidades);

    if (!cidades.remove("Betim")) {
        System.out.println("\nBetim não está no conjunto\n");
    }

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

Saída do programa

remove

```
[João Monlevade, Coronel Fabriciano, Ouro preto, Viçosa, Belo Horizonte]

Betim não está no conjunto

[João Monlevade, Coronel Fabriciano, Ouro preto, Viçosa, Belo Horizonte]
```

Set
contais

1. Para verificar se um elemento pertence ao conjunto basta chamar o método *contains*, retorna um boolean indicando se o elemento está ou não no conjunto;

```
public static void main(String[] args) {

    Set<String> cidades = new HashSet<>();
    String[] nomesCidades = {"João Monlevade", "Coronel Fabriciano",
                             "Belo Horizonte", "Ouro preto", "Viçosa",
                             "João Monlevade"};

    for (String cidade : nomesCidades) {
        cidades.add(cidade);
    }

    if(cidades.contains("João Monlevade")) {
        System.out.println("Cidade cadastrada");
    }
}
```

Set
Percorrendo coleção

1. Elementos de um conjunto não possuem índices como vetores ou ArrayList, se tentarmos acessar diretamente uma determinada posição do HashSet teremos um erro:

```
public static void main(String[] args) {

    Set<String> cidades = new TreeSet<>();
    String[] nomesCidades = {"João Monlevade", "Coronel Fabriciano",
                             "Belo Horizonte", "Ouro preto", "Viçosa",
                             "João Monlevade"};

    for (String cidade : nomesCidades) {
        cidades.add(cidade);
    }

    System.out.println(cidades[0]);
    System.out.println(cidades.get(0));
}
```

Set
Percorrendo coleção

1. Como percorrer um HashSet se um elemento não é identificado pelo seu índice?
   1. Podemos percorrê-lo utilizando o **for aprimorado**;

```
public static void main(String[] args) {
    Set<String> cidades = new HashSet<>();
    String[] nomesCidades = {"João Monlevade", "Coronel Fabriciano",
                             "Belo Horizonte", "Ouro preto", "Viçosa",
                             "João Monlevade"};

    for (String cidade : nomesCidades) {
        cidades.add(cidade);
    }
    System.out.println("**CIDADES CADASTRADAS**");
    for (String cidade : cidades) {
        System.out.println(cidade);
    }
}
```

Saída do programa
percorrendo um conjunto

```
**CIDADES CADASTRADAS**
Belo Horizonte
Coronel Fabriciano
João Monlevade
Ouro preto
Viçosa
```

Iterator
for aprimorado

1. Quando fazemos um for aprimorado, internamente o compilador vai fazer com seja utilizado um **Iterator** para percorrer a coleção;
2. Vide a documentação para mais informações sobre Iterator:

`https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html`

Limpando um conjunto

clear

1. Por fim, podemos remover todos os elementos do conjunto;

```
public static void main(String[] args) {
    Set<String> cidades = new HashSet<>();
    String[] nomesCidades = {"João Monlevade", "Coronel Fabriciano",
                             "Belo Horizonte", "Ouro preto", "Viçosa",
                             "João Monlevade"};

    for (String cidade : nomesCidades) {
        cidades.add(cidade);
    }
    cidades.clear(); // remove todas as cidades
}
```

Collections

Identificando elementos
equals e hashCode

1. Se quisermos inserir ou buscar um elemento dentro de um vetor ou ArrayList basta sabermos a posição para realizarmos a operação;
2. Mas um **HashSet** utiliza uma tabela *hash* para armazenar os dados, como encontrar um elemento?
    1. No Java precisamos de dois métodos para identificar um elemento em uma estrutura que utiliza tabela hash, os métodos **equals** e **hashcode**.

equals e hashCode
Classe Object

1. Os métodos **equals** e **hashCode** são implementados na classe Object, logo, podemos invocá-los com a referência de qualquer objeto;
2. É uma boa prática sempre reescrever os métodos da classe Object.

Tabela Hash
equals e hashCode

1. Cada objeto é classificado pelo seu *hashCode* (obtemos o hashCode de um objeto invocando o método hashCode()), conseguimos agrupar os objetos pelo seu código hash;
2. Se quisermos buscar um elemento dentro do grupo com o mesmo hash, precisamos utilizar o método *equals* (compara se dois objetos são iquais);

Tabela Hash

Analogia: cidades

1. Seguindo o nosso exemplo do armazenamento de cidades, suponha que o código hash de cada cidade seja a primeira letra de seu nome, as cidades seriam agrupadas com outras cidades cujo nome começa com a mesma letra;

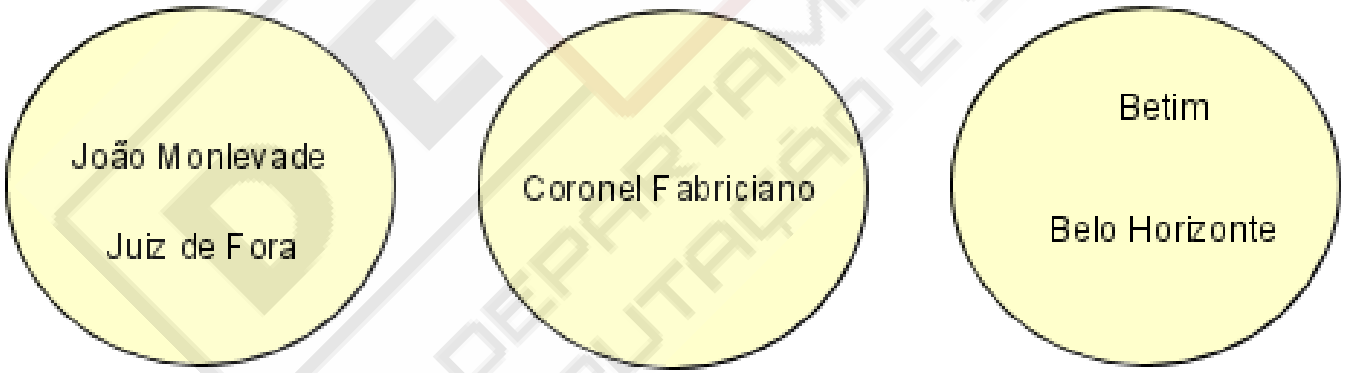


Tabela Hash

Analogia: cidades

1. Se quisermos procurar a cidade de João Monlevade utilizando o método *contains*, o HashSet irá:
   1. Buscar o código hash (letra 'J') para identificar onde procurar a cidade;
      1. Se não existir nenhum cidade com o hash 'J', a cidade que estamos procurando não está na coleção;
      2. Caso exista um conjunto com o hash 'J', o HashSet faz uma chamada ao método *equals* para verificar se a cidade é aquela que estamos procurando.
2. Esta é apenas uma analogia.
3. O método *hashCode* deve retornar um **int**.

equals e hashCode
Utilidade

1. Por que devemos saber trabalhar com os métodos *equals* e *hashCode*?
    1. Todo método que faz comparação de iqualdade entre objetos de coleções, implicitamente faz uma chamada ao método *equals*;
    2. Em nossos exemplos com cidades, não nos preocupamos porque a wrapper class String (assim como as outras classes empacotadoras, Double, Integer...) sobrescreve o *equals* e o *hashCode*. E se as nossas cidades fossem instâncias de uma classe Cidade?

Set
Classe cidade

```
public class Cidade {

    private String nome;
    private int qtdeHabitantes;

    public Cidade(String nome, int qtdeHabitantes) {
        this.nome = nome;
        this.qtdeHabitantes = qtdeHabitantes;
    }

    @Override
    public String toString() {
        return this.nome + " tem " + this.qtdeHabitantes +
                " habitantes";
    }
}
```

Set
Classe de teste

```
public static void main(String[] args) {
    Set<Cidade> cidades = new HashSet<>();

    Cidade jm = new Cidade("João Monlevade", 10000);
    Cidade celFabri = new Cidade("Coronel Fabriciano", 88888);
    Cidade bh = new Cidade("Belo Horizonte", 100000);

    Cidade jm2 = new Cidade("João Monlevade", 10000);

    cidades.add(jm);
    cidades.add(celFabri);
    cidades.add(bh);

    if (cidades.add(jm2)) {
        System.out.println("**João Monlevade foi cadastra**\n");
    }

    for (Cidade c : cidades) {
        System.out.println(c.toString());
    }
}
```

Set
Saida do programa

```
run:
**João Monlevade foi cadastra**

Belo Horizonte tem 100000 habitantes
João Monlevade tem 10000 habitantes
Coronel Fabriciano tem 88888 habitantes
João Monlevade tem 10000 habitantes
BUILD SUCCESSFUL (total time: 0 seconds)
```

Set
equals

1. João Monlevade foi cadastrada duas vezes porque o método *add* faz uma chamada ao método *equals* para verificar se duas cidades são iquais;
    1. O método *equals* implementado na classe Object compara a referência dos objetos;
    2. Se passarmos o objeto *jm* novamente em vez do *jm2* para o método *add* a inserção não será feita, faça o teste.
2. Sobrescrevemos o método equals na classe Cidade, para que o critério de comparação seja o nome da cidade, veremos o resultado...

Set
Classe Cidade

```
public class Cidade {

    private String nome;
    private int qtdeHabitantes;

    public Cidade(String nome, int qtdeHabitantes) {
        this.nome = nome;
        this.qtdeHabitantes = qtdeHabitantes;
    }

    @Override
    public String toString() {
        return this.nome + " tem " + this.qtdeHabitantes +
                " habitantes";
    }

    @Override
    public boolean equals(Object o) {
        Cidade c = (Cidade)o;
        return this.nome.equals(c.getNome());
    }
```

Set
Classe de teste

```
public static void main(String[] args) {
    Set<Cidade> cidades = new HashSet<>();

    Cidade jm = new Cidade("João Monlevade", 10000);
    Cidade celFabri = new Cidade("Coronel Fabriciano", 88888);
    Cidade bh = new Cidade("Belo Horizonte", 100000);

    Cidade jm2 = new Cidade("João Monlevade", 10000);

    cidades.add(jm);
    cidades.add(celFabri);
    cidades.add(bh);

    if (cidades.add(jm2)) {
        System.out.println("**João Monlevade foi cadastra**\n");
    }

    for (Cidade c : cidades) {
        System.out.println(c.toString());
    }
}
```

Set
Saída do programa

```
run:
**João Monlevade foi cadastra**

Belo Horizonte tem 100000 habitantes
João Monlevade tem 10000 habitantes
Coronel Fabriciano tem 88888 habitantes
João Monlevade tem 10000 habitantes
BUILD SUCCESSFUL (total time: 0 seconds)
```

Set
Saída do programa

1. João Monlevade foi inserida duas vezes novamente porque não reescrevemos o método *hashCode*, não sabemos como a classe Object gerou esse código;
    1. logo, não sabemos como as cidades estão sendo agrupadas;
2. Vamos criar o nosso próprio código, agruparemos todas as cidades que começam com mesma letra no mesmo conjunto;
    1. Como o hash precisa ser um **int**, pegaremos o código ASCII do primeiro caracter do nome da cidade;

Set
Classe de teste

```
public class Cidade {

    private String nome;
    private int qtdeHabitantes;

    public Cidade(String nome, int qtdeHabitantes) {
        this.nome = nome;
        this.qtdeHabitantes = qtdeHabitantes;
    }

    @Override
    public String toString() {
        return this.nome + " tem " + this.qtdeHabitantes +
                " habitantes";
    }

    @Override
    public boolean equals(Object o) {
        Cidade c = (Cidade)o;
        return this.nome.equals(c.getNome());
    }

    @Override
    public int hashCode() {
        return this.nome.charAt(0);
    }
```

Set
Saída do programa

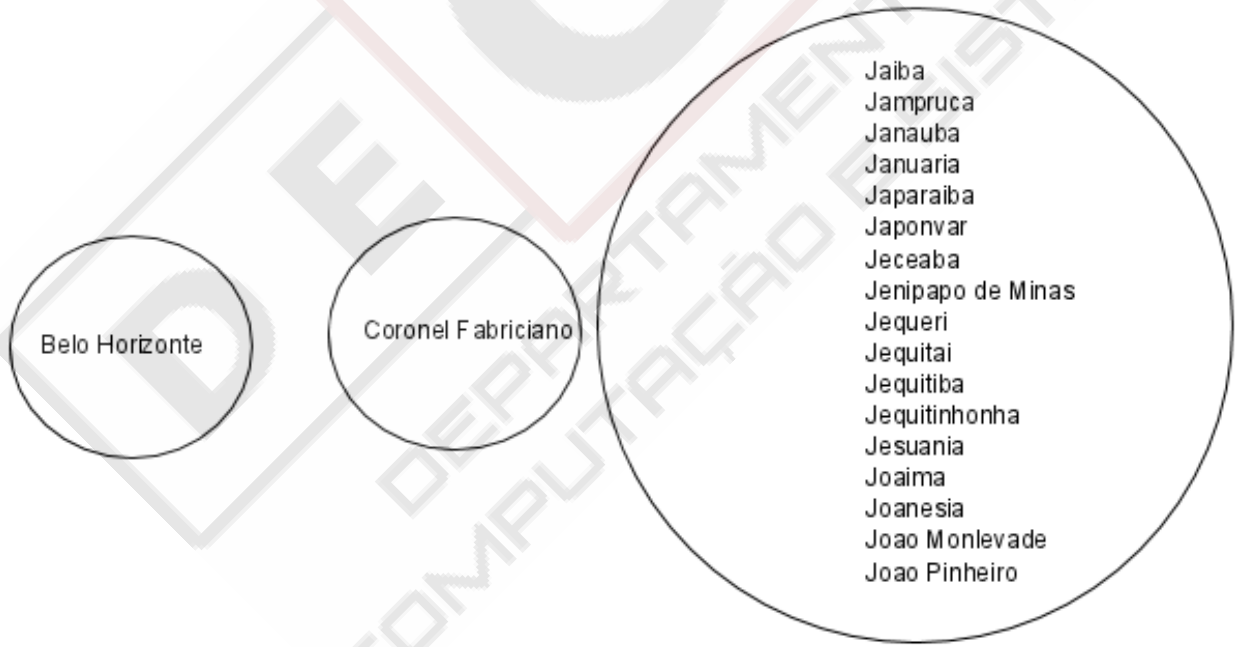
```
Belo Horizonte tem 100000 habitantes
Coronel Fabriciano tem 88888 habitantes
João Monlevade tem 10000 habitantes
BUILD SUCCESSFUL (total time: 0 seconds)
```

Inserção
equals e hashCode

1. A cidade de João Monlevade não foi inserida novamente porque (abstração do algoritmo):
   1. Método *add* faz uma chamada ao método *hashCode* do objeto *jm2*, como o nome começa com 'J', sabemos que essa cidade será inserida no grupo das cidades cujo nome começa com a letra 'J';
   2. Uma vez identificado o grupo, é feita uma chamada ao método *equals*, comparando o atributo nome dos objetos *jm* e *jm2*;
      1. Como os nomes são iquais, o método retorna **true** e o objeto *jm2* não é inserido.

Problema
equals e hashCode

1. Imagine agora que inserimos mais cidades no nosso HashSet, como nosso hashCode é a primeira letra do nome de cada cidade, a distribuição ficaria da seguinte forma:



Problema
equals e HashCode

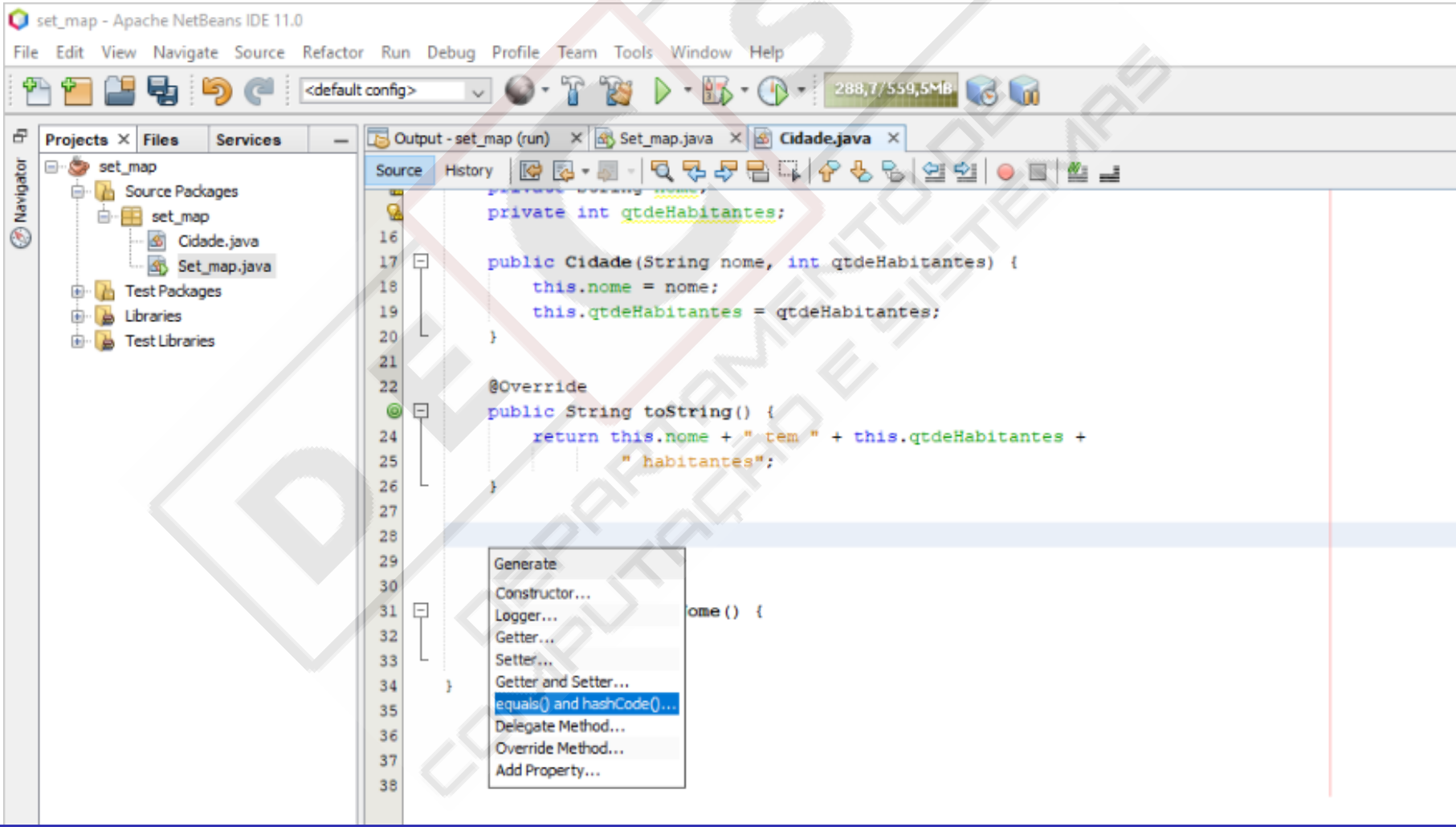
1. Temos um grupo com várias cidades cujo nome começa com a letra 'J';
2. Distribuição está desbalanceada, se quisermos pesquisar por João Monlevade, o método *equals* será chamado para comparar cada uma das cidades do grupo 'J' com a cidade de João Monlevade, até encontrá-la no grupo;
3. O ideal seria se os três grupos de cidades possuíssem a mesma quantidade de elementos.

Gerar código
equals e hashCode

1. Gerar os método equals e hashCode não é uma tarefa trivial, existem regras definidas pela API de JAVA que devem ser seguidas;
2. Felizmente vários ambientes de desenvolvimento como o Netbeans e o Eclipse geram esses métodos, basta escolher para qual(s) atributo(s) queremos gerá-los.

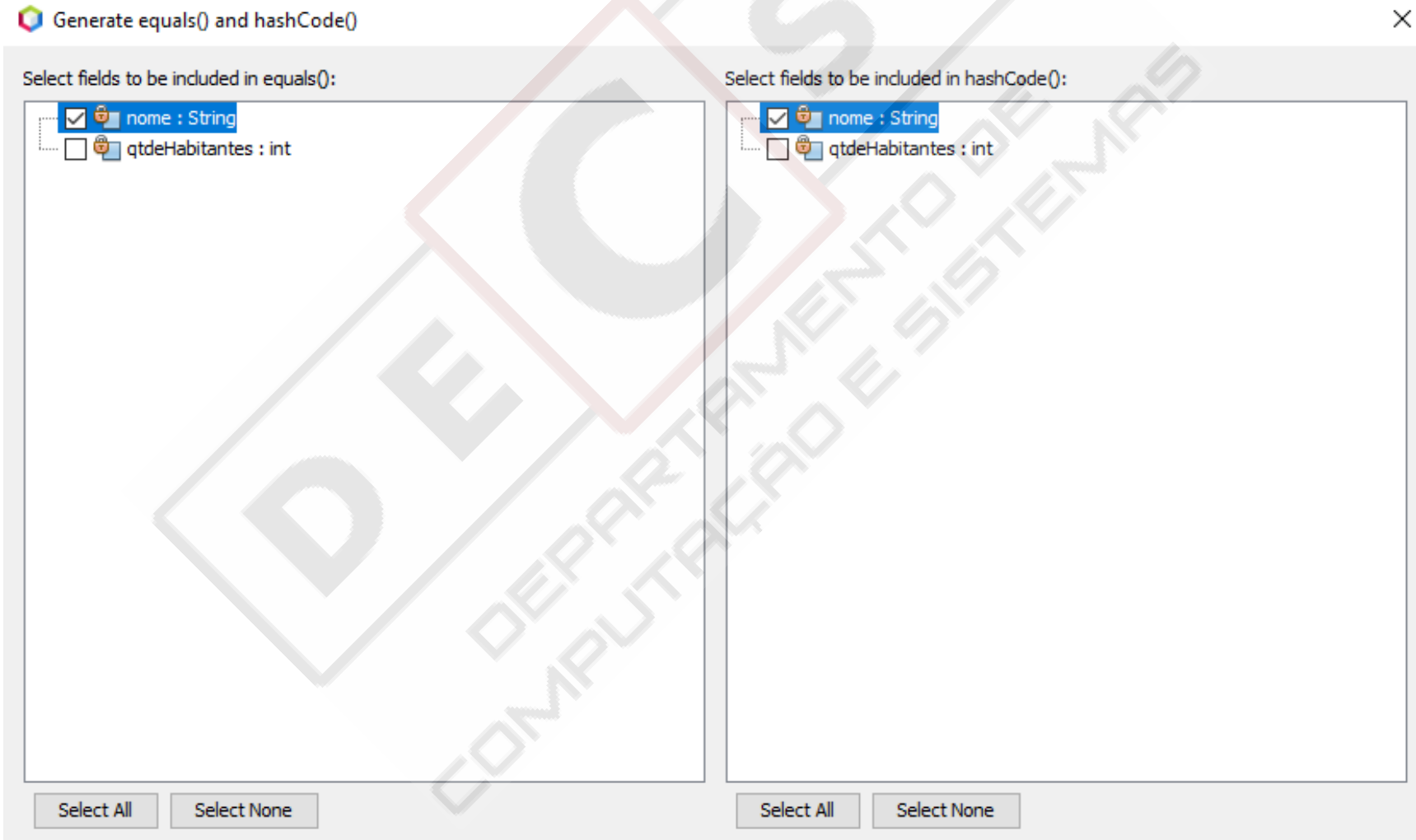
Netbeans

Gerando equals e hashCode



Netbeans

Gerando equals e hashCode



Netbeans
Código gerado

```java
@Override
public int hashCode() {
    int hash = 5;
    hash = 61 * hash + Objects.hashCode(this.nome);
    return hash;
}

@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (obj == null) {
        return false;
    }
    if (getClass() != obj.getClass()) {
        return false;
    }
    final Cidade other = (Cidade) obj;
    if (!Objects.equals(this.nome, other.nome)) {
        return false;
    }
    return true;
}
```

Exercício
[Caelum, 2017]

1. Crie um código que insira 30 mil números numa ArrayList e pesquise-os. Vamos usar um método de System para cronometrar o tempo gasto:
2. ...

Exercício
[Caelum, 2017]

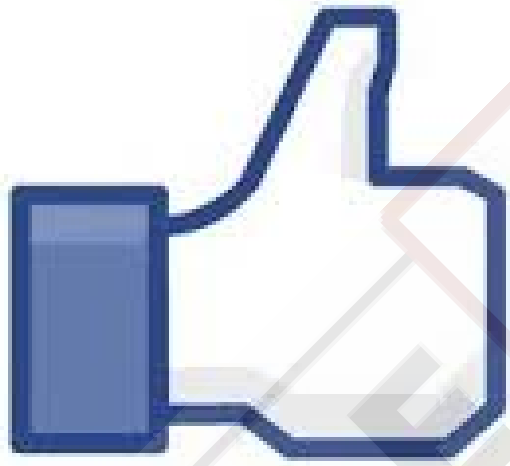
```
public class TestaPerformance {

    public static void main(String[] args) {
        System.out.println("Iniciando...");
        Collection<Integer> teste = new ArrayList<>();
        long inicio = System.currentTimeMillis();

        int total = 30000;

        for (int i = 0; i < total; i++) {
            teste.add(i);
        }

        for (int i = 0; i < total; i++) {
            teste.contains(i);
        }

        long fim = System.currentTimeMillis();
        long tempo = fim - inicio;
        System.out.println("Tempo gasto: " + tempo);
    }
}
```

Exercício
[Caelum, 2017]

1. Troque a ArrayList por um HashSet e verifique o tempo que vai demorar:

```
Collection<Integer> teste = new HashSet<>();
```

1. O que é lento? A inserção de 30 mil elementos ou as 30 mil buscas? Descubra computando o tempo gasto em cada for separadamente.



MUITO OBRIGADO!!!

Referências Bibliográficas I

Caelum (2017).
Java e orientação a objeto.
https://www.caelum.com.br/download/caelum-java-objetos-fj11.pdf.