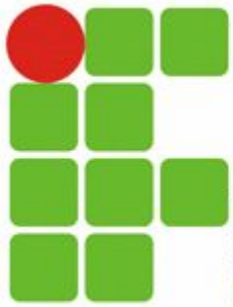


INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
FLUMINENSE

Interface SET (java.util.Set)

Disciplina
PROGRAMAÇÃO ORIENTADA A OBJETOS

Prof. Roberta B Tôrres

Conjunto (java.util.Set)

Um conjunto (**Set**) funciona de forma análoga aos conjuntos da matemática. É uma coleção que não permite elementos duplicados.

É um subtipo de ***Collection*** que representa uma coleção não-indexada de objetos.

Uma outra característica fundamental de Set é o fato de que a ordem em que os elementos são armazenados pode não ser a ordem na qual eles foram inseridos no conjunto.

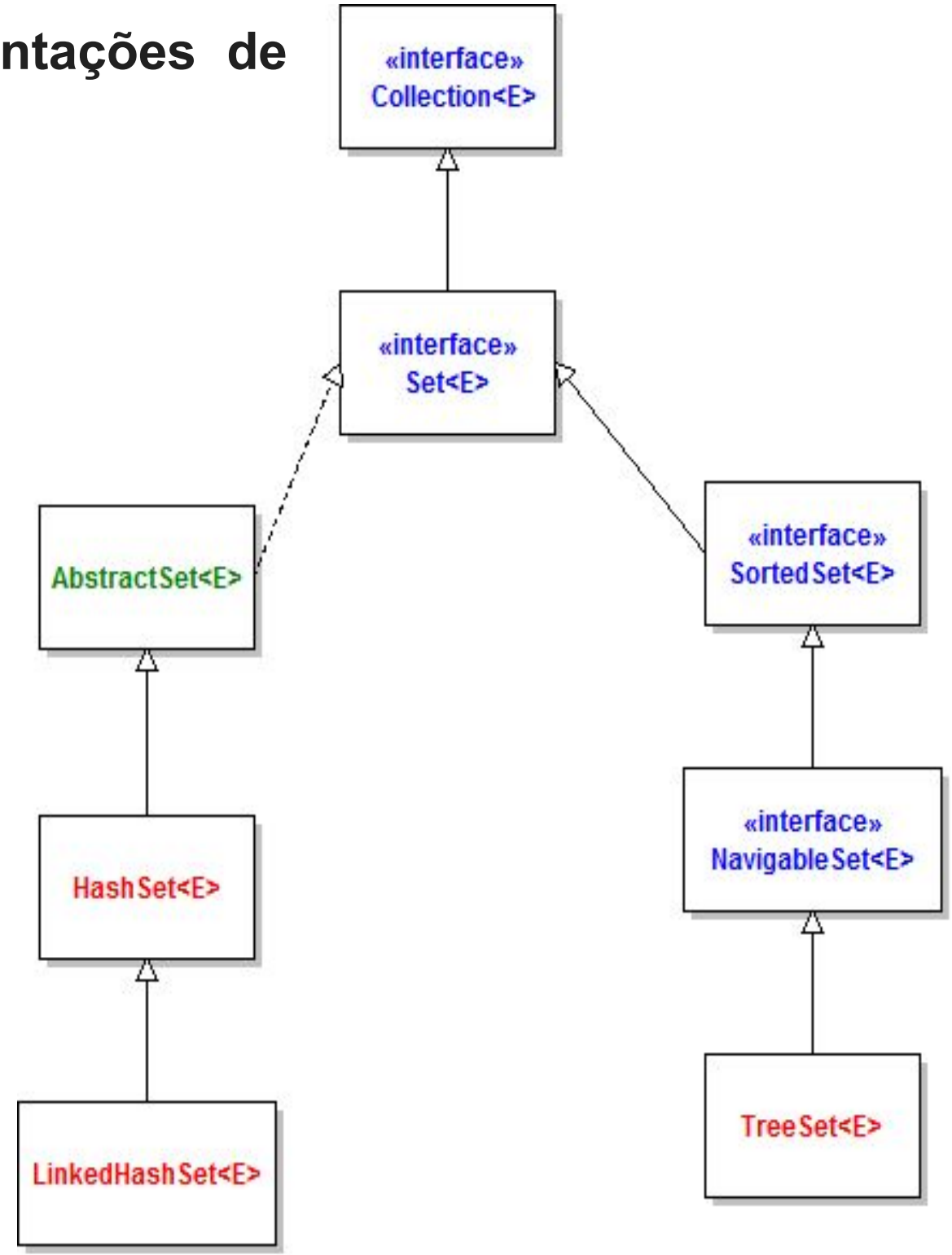
A interface não define como deve ser este comportamento. Tal ordem varia de implementação para implementação.

As principais implementações de SET são as classes:

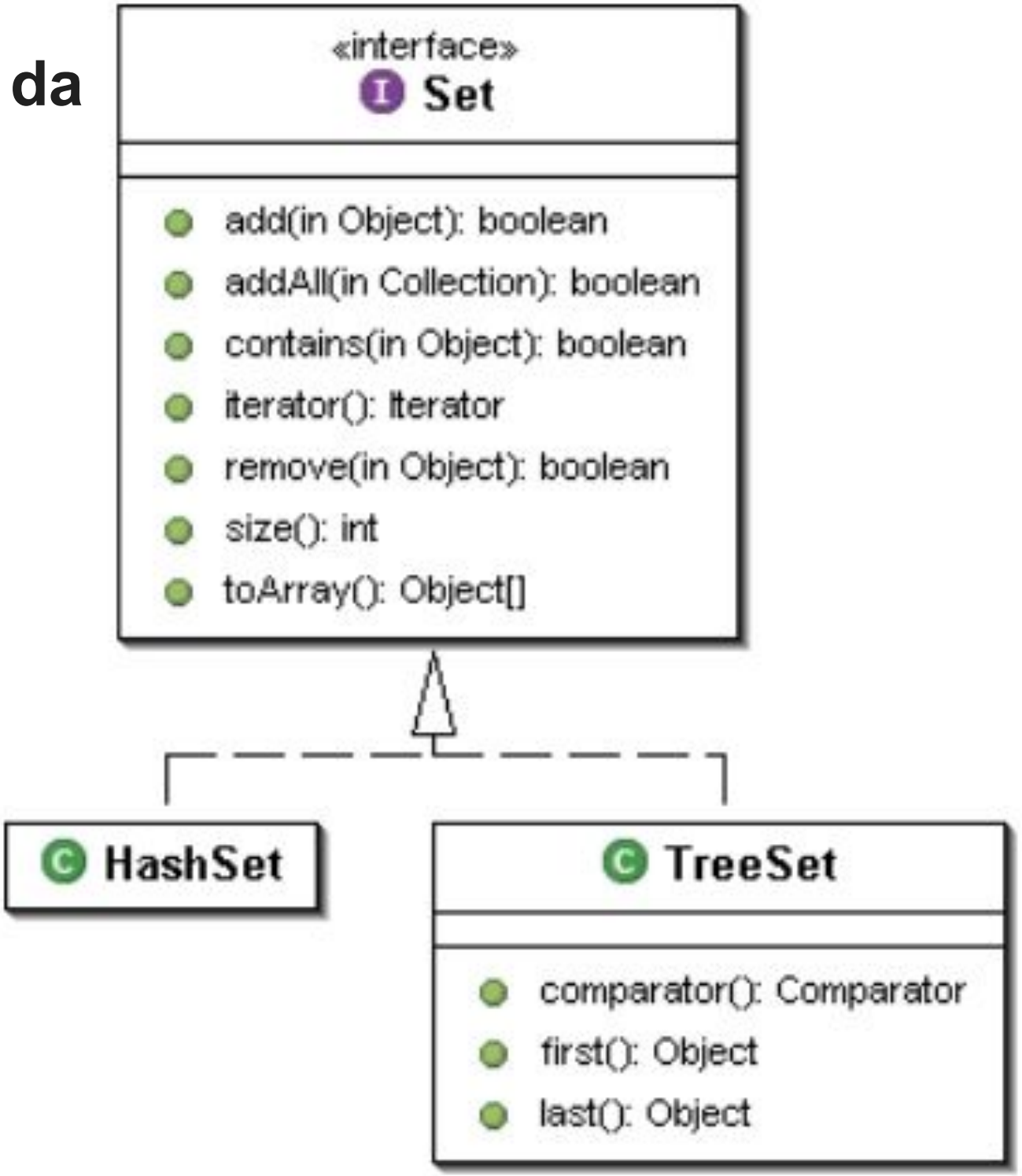
- **HashSet**
- **LinkedHashSet**
- **TreeSet**

Essas 3 classes implementam a interface **SET**, logo, temos os mesmos métodos para as 3 estruturas.

O que difere em cada uma é a forma com que é implementado o algoritmo.



Alguns métodos da interface Set.



Implementações de SET

- HashSet
  - Armazena cada um de seus elementos em um espaço de memória sempre utilizando os métodos equals() e hashCode() do objeto inserido para comparação com cada um dos objetos já existentes no set.
  - Trata-se de uma das coleções mais eficientes de todo o framework Java.
- LinkedHashSet
  - Implementação da interface Set que armazena seus elementos na mesma ordem em que foram inseridos.

SET: Exemplos

```
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;

public class DevmediaSet {

    public static void main(String[] args) {

        Set<Carro> carros;

        //Você pode implementar com HashSet
        carros = new HashSet<Carro>();

        //Pode também optar pelo LinkedHashSet
        carros = new LinkedHashSet<Carro>();
    }
```

SET: Exemplos

O código a seguir cria um conjunto e adiciona diversos elementos, e alguns repetidos:

```
Set<String> cargos = new HashSet<>();

cargos.add("Gerente");
cargos.add("Diretor");
cargos.add("Presidente");
cargos.add("Secretária");
cargos.add("Funcionário");
cargos.add("Diretor"); // repetido!

// imprime na tela todos os elementos
System.out.println(cargos);
```

"Diretor" é inserido apenas uma vez, já que este tipo de coleção não permite elementos duplicados.

SET: Exemplos

```
Set<Funcionario> conjunto = new HashSet<Funcionario>();

conjunto.add(new Funcionario(203, “Maria”));
conjunto.add(new Funcionario(112, “Manuel”));
conjunto.add(new Funcionario(205, “Joaquim”));
conjunto.add(new Funcionario(185, “Maria”));

System.out.println(“Quantidade de funcionarios: ” +
        conjunto.size());
```

HashSet

HashSet é a coleção mais eficiente de todas, mas não garante a ordenação dos seus elementos. Internamente faz uso de HashTable em sua implementação para armazenar e fornecer a garantia de seus objetos não se repetirem.

A coleção **HashSet** não garante a ordem de inserção, podendo embaralhar os objetos quando for percorrida. A garantia de continuidade na ordem dos elementos inseridos é zero, ou seja, esse tipo de estrutura é indicada se você precisa apenas garantir a alta performance sem se importar com a ordem com que os elementos estão ordenados.

A complexidade desta estrutura é **O(1)**, ou seja, não importa o quanto se adicione, remova, retire, entre outras operações, o tempo de execução sempre será o mesmo.

E isso é extremamente crítico em processos onde temos uma situação crítica com milhões de dados a serem inseridos em um Set.

HashSet: Exemplo

```
HashSet<Dog> dset = new HashSet<Dog>();
    dset.add(new Dog(2));
    dset.add(new Dog(1));
    dset.add(new Dog(3));
    dset.add(new Dog(5));
    dset.add(new Dog(4));
    Iterator<Dog> iterator = dset.iterator();
    while (iterator.hasNext()) {
        System.out.print(iterator.next() + " ");

Saída: 5 3 2 1 4
```

LinkedHashSet

A classe **LinkedHashSet** é muito semelhante a **HashSet**, também não permite elementos duplicados.

A diferença em **LinkedHashSet** é que os elementos continuam na ordem que foram inseridos. Esta estrutura utiliza uma lista duplamente encadeada para garantir, que ao ser percorrida, a ordem de inserção dos elementos será respeitada.

Portanto, tendo em mente que duplicatas não são permitidas e a ordem de inserção é preservada, **LinkedHashSet** é amplamente utilizado na construção de aplicativos baseados em cache.

LinkedHashSet: Exemplo

```
public class LinkedHashSetExample {

    public static void main(String[] args) {

        int[] a = {1, 4, 1, 3, 7, 3, 4};
        Set<Integer> s = new LinkedHashSet<Integer>();

        for(int i = 0; i < a.length; i++) {
            s.add(a[i]);
        }

        Iterator<Integer> i = s.iterator();
        while(i.hasNext()) {
            System.out.print(i.next() + " ");
        }
    }
}
```

Saída do Programa

```
LinkedHashSetExampl
1 4 3 7
```

LinkedHashSet: Exemplo

```
LinkedHashSet<Dog> dset = new LinkedHashSet<Dog>();
dset.add(new Dog(2));
dset.add(new Dog(1));
dset.add(new Dog(3));
dset.add(new Dog(5));
dset.add(new Dog(4));
Iterator<Dog> iterator = dset.iterator();
while (iterator.hasNext()) {
    System.out.print(iterator.next() + " ");
}

Saída: 2 1 3 5 4
```

TreeSet

TreeSet implementa um algoritmo conhecido por red-black tree (árvore rubro-negra).

É a única classe, da hierarquia Set, que implementa a interface **SortedSet** em vez de **Set** diretamente (mas de qualquer forma SortedSet implementa Set, logo continuamos tendo os mesmo métodos no TreeSet).

Pelo fato de ele implementar **SortedSet**, seus elementos são ordenados automaticamente, ou seja, independente da ordem que você inserir os elementos, eles serão ordenados.

Mas isso tem um custo, a complexidade para os métodos **add**, **remove** e **contains é O(log (n))**, bem maiores que do **HashSet.** O(log (n)) não é bem uma complexidade exponencial, mas é bem maior que a complexidade de HashSet (que é O(1) e tem seu tempo inalterado).

Por implementar SortedSet o TreeSet oferece mais alguns métodos como: first(), last(), headSet(), tailSet() e outros.

TreeSet: Exemplo

```
TreeSet<Integer> tree = new TreeSet<Integer>();
    tree.add(12);
    tree.add(63);
    tree.add(34);
    tree.add(45);

    Iterator<Integer> iterator = tree.iterator();
    System.out.print("Tree set data: ");
    while (iterator.hasNext()) {
        System.out.print(iterator.next() + " ");
    }

Saída: 12 34 45 63
```

TreeSet: Ordenação de Objetos

No exemplo anterior, **TreeSet** opera sobre tipos primitivos, no caso números inteiros.

Mas pensando em Objetos, como ele saberia como ordená-los?

```
public class TestTreeSet {
    public static void main(String[] args) {
        TreeSet<Dog> dset = new TreeSet<Dog>();
        dset.add(new Dog(2));
        dset.add(new Dog(1));
        dset.add(new Dog(3));

        Iterator<Dog> iterator = dset.iterator();

        while (iterator.hasNext()) {
            System.out.print(iterator.next() + " ");
        }
    }
}
```

O programa ao lado, por exemplo, dará um erro em tempo de execução:

Exception in thread “main” java.lang.ClassCastException: collection.Dog cannot be cast to java.lang.Comparable.

TreeSet: Ordenação de Objetos

O erro (slide anterior) ocorre por um fato simples: Como o **TreeSet** vai ordenar uma lista de objetos Dog's, se não dissermos a ele por onde ordenar ?

A solução é implementar a interface **Comparable** e o método "**compareTo**". É através deste método que diremos como o **TreeSet** deve ordenar nosso Objeto em questão.

TreeSet: Ordenação de Objetos

- *Para ordenar objetos é preciso compará-los.*
- *Como estabelecer os critérios de comparação?*
  - *equals() apenas informa se um objeto é igual a outro, mas não informa se "é maior" ou "menor"*
- *Solução: interface java.lang.Comparable*
  - *Método a implementar:*

```
public int compareTo(Object obj);
```

- *Para implementar, retorne*
  - *Um inteiro menor que zero se objeto atual for "menor" que o recebido como parâmetro*
  - *Um inteiro maior que zero se objeto atual for "maior" que o recebido como parâmetro*
  - *Zero se objetos forem iguais*

TreeSet: Ordenação de Objetos

```
class Dog implements Comparable<Dog>{
    int size;

    public Dog(int s) {
        size = s;
    }

    public String toString() {
        return size + "";
    }

    @Override
    public int compareTo(Dog o) {
            return size - o.size;
    }
}
```

Agora sim, podemos usar o **TreeSet** que será ordenado automaticamente, e a saída será: 1 2 3.

Exemplo: java.lang.Comparable

```
public class Coisa implements Comparable {
  private int id;
  public Coisa(int id) {
    this.id = id;
  }
  public int compareTo(Object obj) {
    Coisa outra = (Coisa) obj;
    if (id >  outra.id) return 1;
    if (id <  outra.id) return -1;
    if (id == outra.id) return 0;
  }
}
```

- *Como usar*

```
Coisa c1 = new Coisa(123);
Coisa c2 = new Coisa(456);
if (c1.compareTo(c2)==0) System.out.println("igual");
Coisa coisas[] = {c2, c1, new Coisa(3)};
Arrays.sort(coisas);
```

Usa compareTo()

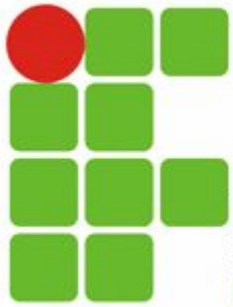
SET
Vantagens & Desvantagens

A principal diferença entre as implementações da interface **Set** está na velocidade e ordenação que elas proporcionam

O uso de **Set** pode parecer desvantajoso, já que não armazena a ordem (dependendo da implementação) e não aceita elementos repetidos.

Não há métodos que trabalham com índices, como o get(int) que as listas possuem.

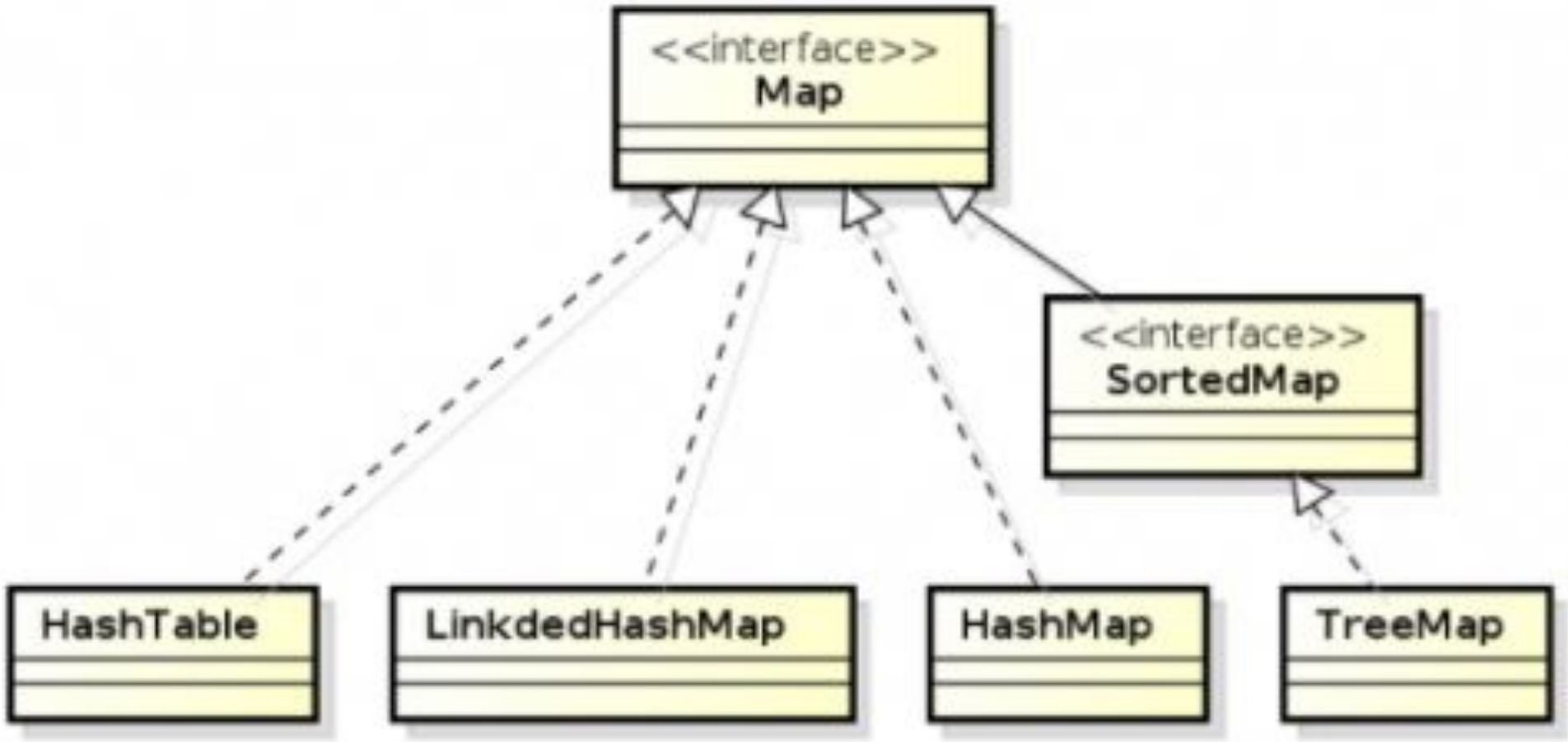
A grande vantagem do **Set** é que existem implementações, como a **HashSet**, que possui uma performance incomparável com as **Lists** quando usado para pesquisa (método **contains** por exemplo).



INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
FLUMINENSE

Interface MAP e suas implementações

Hierarquia de Map



Mapas (java.util.Map)

Muitas vezes queremos buscar rapidamente um objeto dado alguma informação sobre ele.

Por exemplo: dada a placa do carro, obter todos os dados do carro. Podemos utilizar uma lista para isso e percorrer todos os seus elementos, mas isso pode ser péssimo para a performance, mesmo para listas não muito grandes. Aqui entra a estrutura MAP.

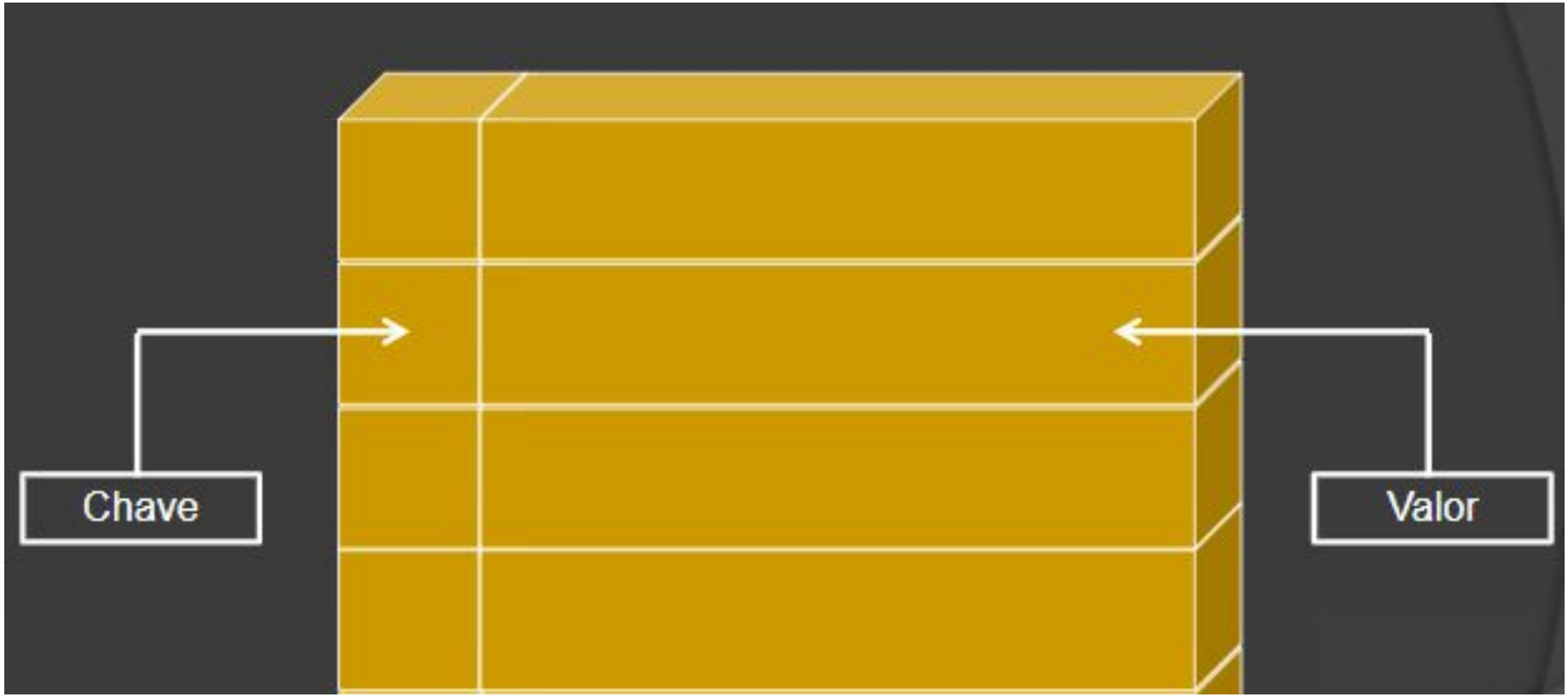
Um **mapa** é composto por um conjunto de associações entre um **objeto chave** a um **objeto valor**.

Equivale ao conceito de dicionário, usado em várias linguagens. Algumas linguagens, como Perl ou PHP, possuem um suporte mais direto a mapas, onde são conhecidos como matrizes/arrays associativas.

Interface Map

- Objetos Map são semelhantes a vetores mas, em vez de índices numéricos, usam chaves, que são objetos
  - Chaves são unívocas (um Set)
  - Valores podem ser duplicados (um Collection)
- Métodos
  - void put(Object key, Object value): acrescenta um objeto
  - Object get (Object key): recupera um objeto
  - Set keySet(): retorna um Set de chaves
  - Collection values(): retorna um Collection de valores
  - Set entrySet(): retorna um set de pares chave-valor contendo objetos representados pela classe interna Map.Entry

Interface Map



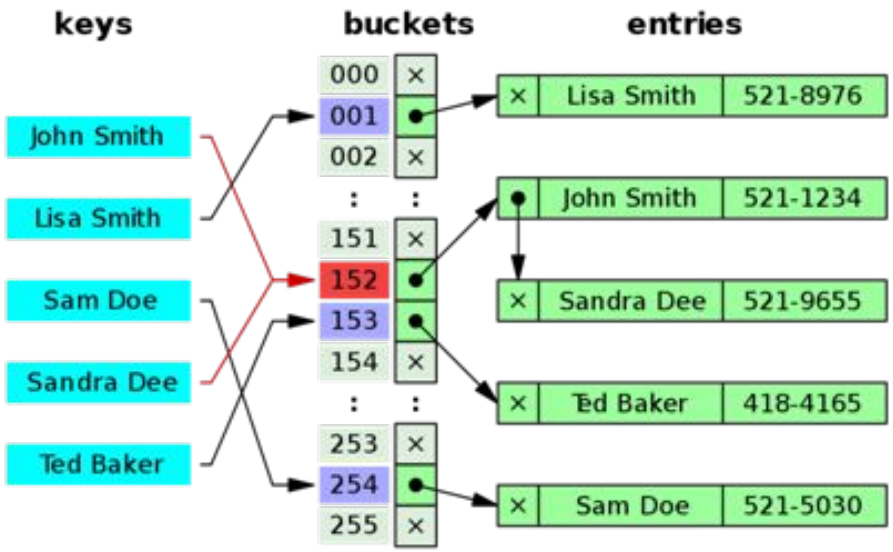
java.util.Map é um mapa, sendo possível mapear uma chave a um valor. Por exemplo: mapeie à chave "empresa" ao valor "Caelum"; ou mapear à chave "rua" ao valor "Vergueiro". Semelhante a associações de palavras que podemos fazer em um dicionário.

Interface Map

Os objetos “**Map**” confiam seus dados em um algoritmo **hash** (hash code). Esse algoritmo transforma uma grande quantidade de dados em uma pequena quantidade de informações, sendo que o mecanismo de busca se baseia na construção de índices.

Um exemplo prático pode ser usado como uma lista telefônica onde a letra seria o índice a ser procurado, para conseguir achar mais fácil o nome desejado.

Essa interface é um objeto que mapeia valores para chaves, ou seja, através da chave consegue ser acessado o valor configurado, sendo que a chave não pode ser repetida ao contrário do valor, mas se caso tiver uma chave repetida é sobrescrito pela última chamada. Faz parte do pacote java.util e não possui métodos da interface Collection.

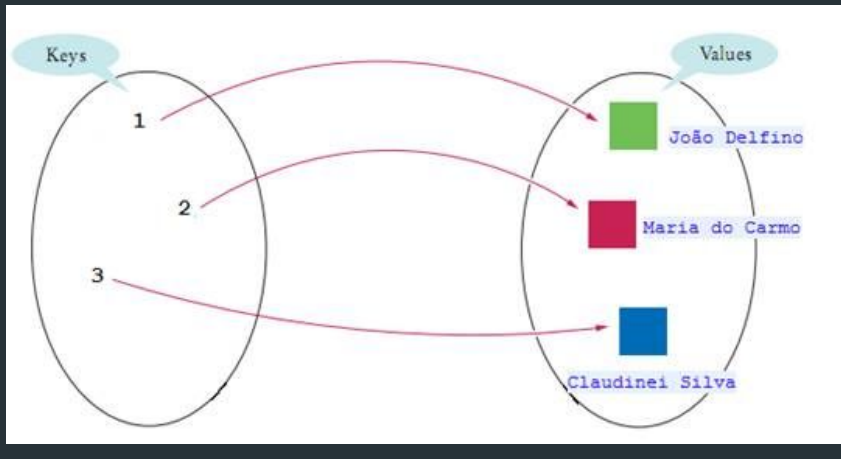


Implementações de Map e Map.Entry

- *HashMap*
  - *Escolha natural quando for necessário um vetor associativo*
  - *Acesso rápido: usa Object.hashCode() para organizar e localizar objetos*
- *TreeMap*
  - *Mapa ordenado*
  - *Contém métodos para manipular elementos ordenados*
- *Map.Entry*
  - *Classe interna usada para manter pares chave-valor em qualquer implementação de Map*
  - *Principais métodos: Object getKey(), retorna a chave do par; Object getValue(), retorna o valor.*

Map - Exemplo

```java
import java.util.HashMap;
import java.util.Map;

public class TestaInterfaceMap {

  public static void main(String[] args) {

    Map<integer, string=""> mapaNomes = new HashMap<integer, string="">();
    mapaNomes.put(1, "João Delfino");
    mapaNomes.put(2, "Maria do Carmo");
    mapaNomes.put(3, "Claudinei Silva");

    System.out.println(mapaNomes);

    //resgatando o nome da posição 2
    System.out.println(mapaNomes.get(2));

  }
}
```



Map - Exemplo

```
Map map = new HashMap();
map.put("um", new Coisa("um"));
map.put("dois", new Coisa("dois"));
(...)
Set chaves = map.keySet();
Collection valores = map.values();
(...)
Coisa c = (Coisa) map.get("dois");
(...)
Set pares = map.entrySet();
Iterator entries = pares.iterator();
Map.Entry one = entries.next();
String chaveOne = (String)one.getKey();
Coisa valueOne = (Coisa)one.getValue();
```

Classe HashMap

Essa classe é a implementação da interface Map mais trabalhada no campo de desenvolvimento.

Características:

- Os elementos não são ordenados;
- É rápida na busca/inserção de dados;
- Permite inserir valores e chaves nulas;

Para interagir sobre um mapa é preciso trabalhar com a interface ***Collection*** ou métodos **set()** para converter esse mapa em um conjunto de dados.

Classe HashTable

Essa classe trabalha com algoritmo **hash** para conversão das chaves e um mecanismo de pesquisa de valores, sendo que tem seus métodos sincronizados (*thread-safe*) que permitem checar acessos concorrentes e armazenagem.

Também possui uma eficiente pesquisa de elementos baseados em chave-valor, mas não aceita valores nulos.

HashTable - Exemplo

```java
class Cliente{
  public String cpf;
  public String nome;

  public Cliente(String cpf, String nome) {
    this.cpf = cpf;
    this.nome = nome;
  }

  @Override
  public String toString() {
    return cpf + " | " + nome;
  }
}
```

Exemplo de como pode ser desenvolvido um mapa com objeto “Cliente” por HashTable.

```java
import java.util.Hashtable;

public class TesteHashTable {

  public static void main(String[] args) {

    Cliente c1 = new Cliente("754.856.869-88","Valdinei Santos");
    Cliente c2 = new Cliente("828.929.222.12","Claire Moura");
    Cliente c3 = new Cliente("156.565.556-88","Vagner Seller");

    Hashtable<integer, cliente=""> ht = new Hashtable<integer, cliente="">();
    ht.put(1, c1);
    ht.put(2, c2);
    ht.put(3, c3);

    System.out.println("CPF \t\t Cliente");
    for (int i = 1; i <= ht.size(); i++) {
      System.out.println(ht.get(i));
    }
  }
}
```

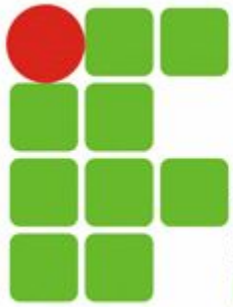
CURIOSIDADE

Entenda como funciona a estrutura de dados

Tabela hash ou Tabela de dispersão ou Tabela de espalhamento

- **https://blog.pantuza.com/artigos/tipos-abstratos-de-dados-tabela-hash**

- **https://www.techtudo.com.br/artigos/noticia/2012/07/o-que-e-hash.html**



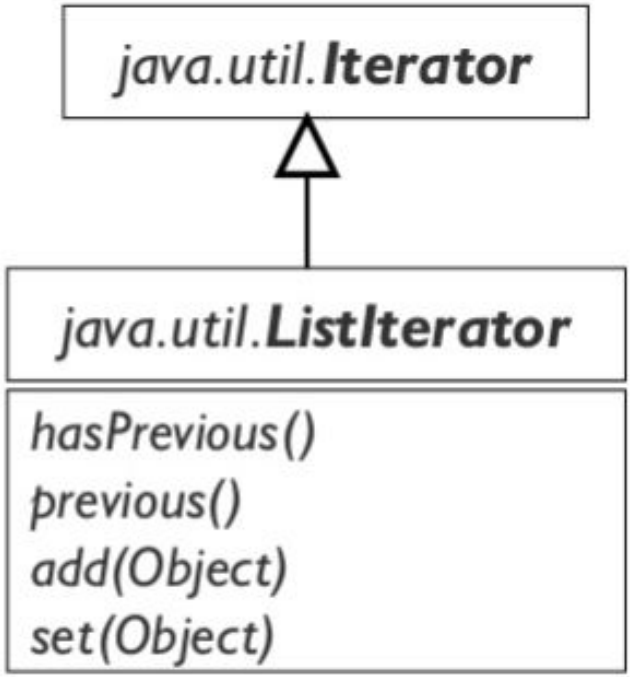
INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
FLUMINENSE

Interface Iterator (java.util.Iterator)

Iterator

- *Para navegar dentro de uma Collection e selecionar cada objeto em determinada seqüência*
  - *Uma coleção pode ter vários Iterators*
  - *Isola o tipo da Coleção do resto da aplicação*
  - *Método iterator() (de Collection) retorna Iterator*

```
package java.util;
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```



- *ListIterator possui mais métodos*
  - *Método listIterator() de List retorna ListIterator*

Material Complementar

- **https://slideplayer.com.br/slide/10074022/ - SET e MAP**
- **https://www.devmedia.com.br/diferencas-entre-treeset-hashset-e-linkedhashset-em-java/29077**
- https://www.devmedia.com.br/classes-stack-queue-e-hashtable-colecoes-estrutura-da-linguagem-parte-3/19256
- https://www.devmedia.com.br/overview-of-java-arraylist-hashtable-hashmap-hashet-linkedlist/30383
- https://www.devmedia.com.br/java-collections-set-list-e-iterator/29637
- https://slidetodoc.com/ine-5408-estruturas-de-dados-6-3-2/ (explicação de árvore rubro negra)
- https://pt.slideshare.net/denistuning/java-14
- https://www.devmedia.com.br/conhecendo-a-interface-map-do-java/37463