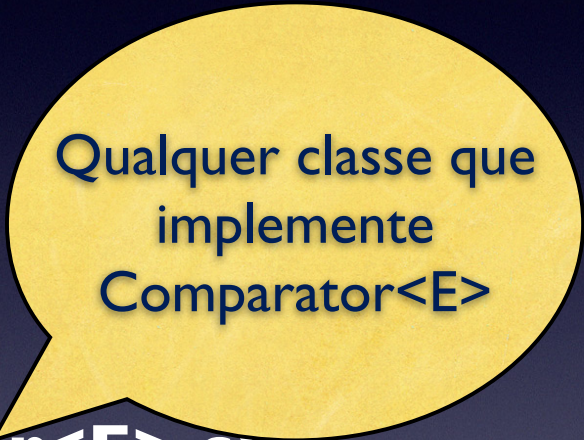


# TreeSet<E>

## Construtores

- **public TreeSet<E>()**
  - Utiliza ordem natural de **E**
- **public TreeSet<E>(Comparator<E> c)**
  - Utiliza o comparator **c** para ordenar os objectos dentro do conjunto



Qualquer classe que implemente **Comparator<E>**



# Comparator<E>

- Permitem definir diferentes critérios de ordenação
- Implementam o método **int compare(E e1, E e2)**
  - Mesmas regras de **compareTo** aplicadas a **e1** e **e2**

```
/**
 * Comparator de Aluno - ordenação por número.
 *
 * @author José Creissac Campos
 * @version 20160403
 */

import java.util.Comparator;
public class ComparatorAlunoNum implements Comparator<Aluno> {
    public int compare(Aluno a1, Aluno a2) {
        int n1 = a1.getNumero();
        int n2 = a2.getNumero();

        if (n1==n2) return 0;
        if (n1>n2) return 1;
        return -1;
    }
}
```

```
/**
 * Comparator de Aluno - ordenação por nome.
 *
 * @author José Creissac Campos
 * @version 20160403
 */

import java.util.Comparator;
public class ComparatorAlunoNome implements Comparator<Aluno> {
    public int compare(Aluno a1, Aluno a2) {
        return a1.getNome().compareTo(a2.getNome());
    }
}
```



# Interfaces

- **Comparable<T>** e **Comparator<T>** são *interfaces*
- Interfaces definem APIs (conjunto de métodos) que as classes que as implementam devem codificar (associar um comportamento)
- Interfaces definem novos Tipos de Dados

# Interfaces Comparable e Comparator

## Interface Comparable<T>

### Method Summary

All Methods

Instance Methods

Abstract Methods

Modifier and Type

Method and Description

int

`compareTo(T o)`

Compares this object with the specified object for order.

## Interface Comparator<T>

### Method Summary

All Methods

Static Methods

Instance Methods

Abstract Methods

Default Methods

Modifier and Type

Method and Description

int

`compare(T o1, T o2)`

Compares its two arguments for order.

boolean

`equals(Object obj)`

Indicates whether some other object is "equal to" this comparator.



# Comparators como expressão lambda

- Os comparators também podem ser definidos como um lambda ou como uma classe anónima.
- Ao utilizar as expressões lambda para fornecer o algoritmo de comparação evita-se o trabalho de ter de criar um objecto para conter um método (neste caso o método `compare`)



# Criação de estruturas ordenadas

- Criar um `TreeSet<Aluno>` com a comparação dada pela ordem natural:

```
TreeSet<Aluno> turma = new TreeSet<>();
```

- Criar um `TreeSet` de `Aluno` com ordenação por comparador

```
TreeSet<Aluno> alunos = new TreeSet<>(new ComparatorAlunoNome());
```



- Criar um TreeSet definido o comparator do mesmo na invocação (via classe anónima).  
Excessivamente complicado!

```
TreeSet<Aluno> teóricas = new TreeSet<>(  
    new Comparator<Aluno>() {  
        public int compare(Aluno a1, Aluno a2) {  
            return a1.getNome().compareTo(a2.getNome());  
        }  
    });
```

- Esta declaração corresponde a uma classe anónima interna, que não existe nas classes visíveis no projecto e só é utilizada para este parâmetro.

- Uma outra forma é recorrer a um método anónimo, escrito sob a forma de uma expressão lambda.

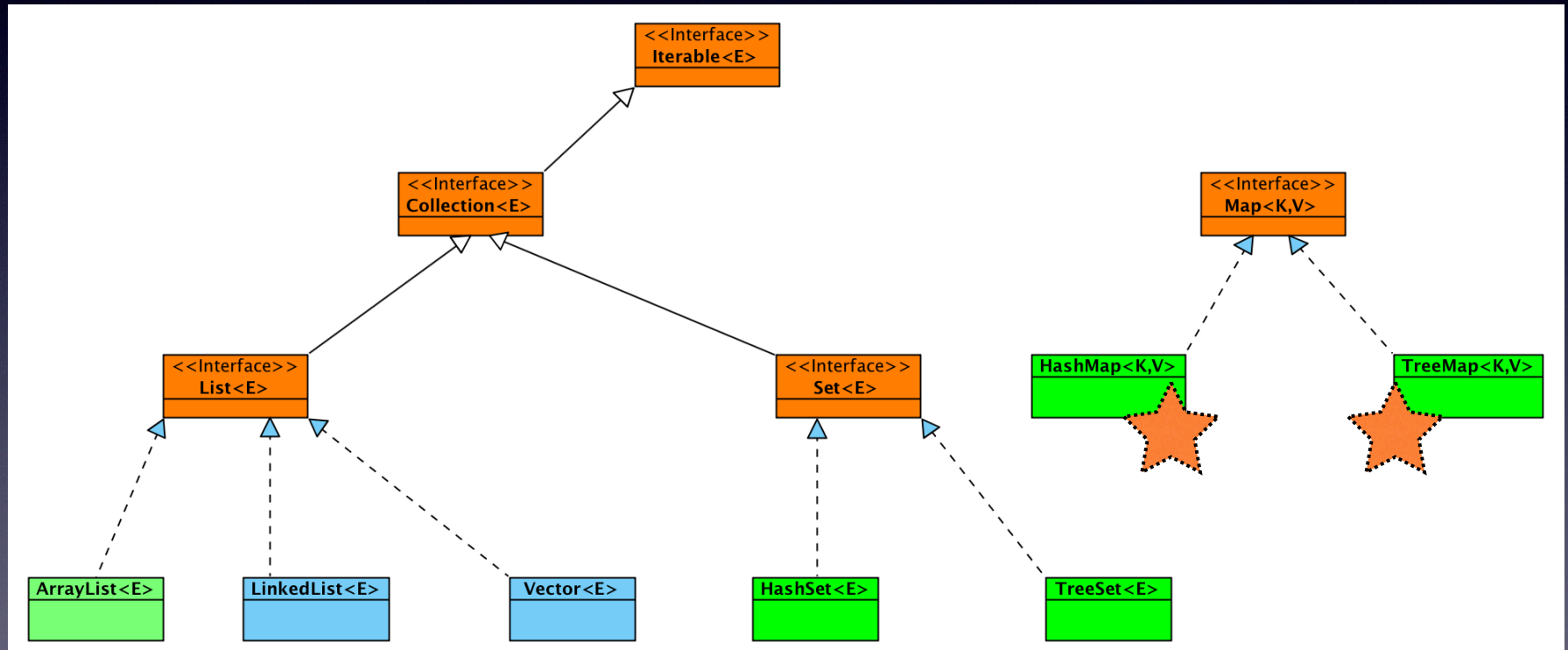
```
TreeSet<Aluno> praticas = new TreeSet<>((a1,a2) ->  
                                         a1.getNome().compareTo(a2.getNome()));
```

- ou, se quisermos reutilizar as expressões:

```
Comparator<Aluno> comparador = (a1, a2) -> a1.getNome().compareTo(a2.getNome());  
TreeSet<Aluno> tutorias = new TreeSet<>(comparador);
```



# Coleções e Maps

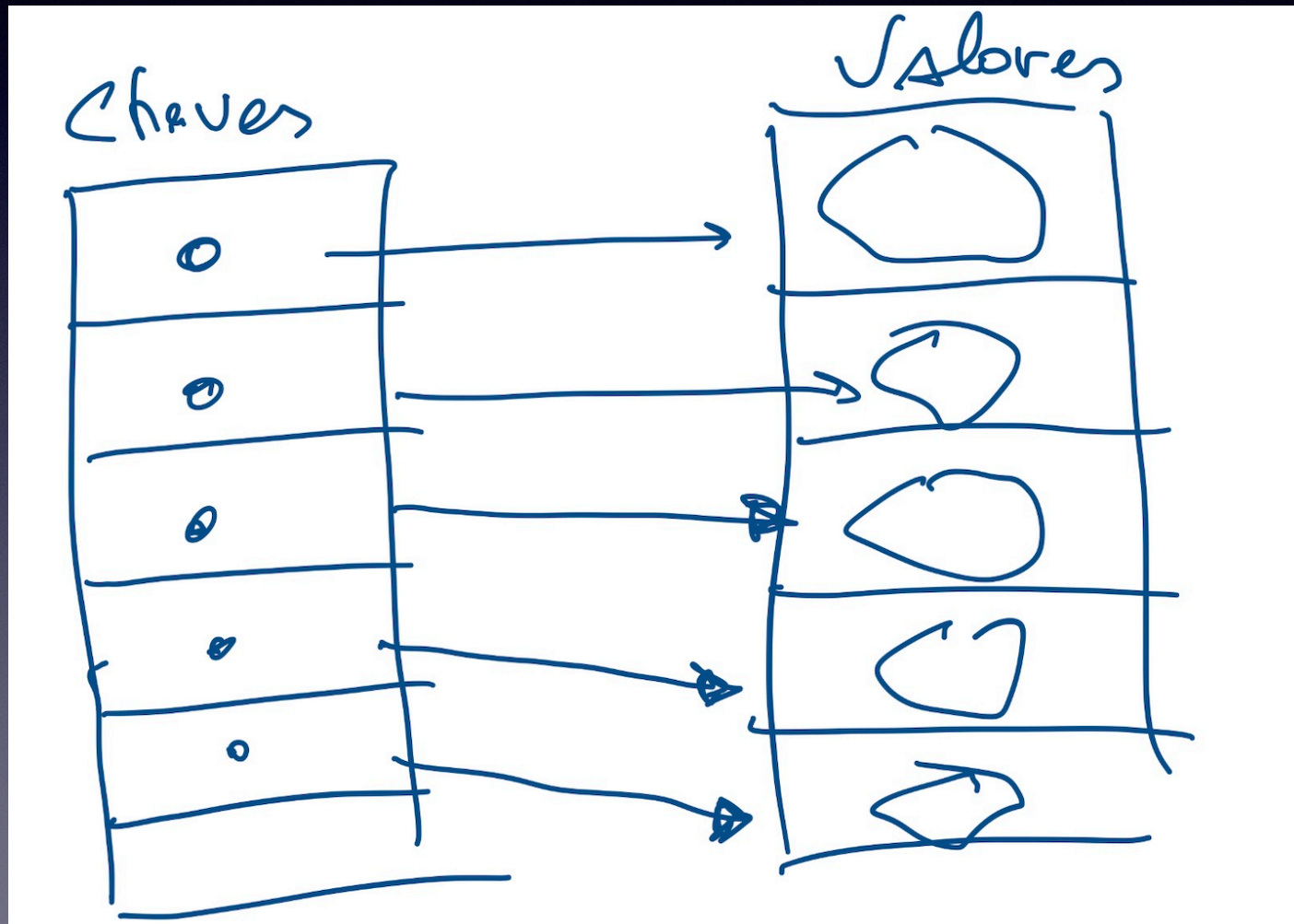


# Map<K,V>

- Quando se pretende ter uma associação de um objecto chave a um objecto valor
- Na dimensão das chaves não existem elementos repetidos (é um conjunto!)
- Duas implementações disponíveis:  
**HashMap<K,V>** e **TreeMap<K,V>**
- aplicam-se à dimensão das chaves as considerações anteriores sobre conjuntos



# Map<K,V>



# Map<K,V>

Adicionar elementos	<code>boolean put(K key,V value)</code> <code>boolean putAll(Map m)</code> <code>V putIfAbsent(K key,V value)</code>
Alterar o Map	<code>void clear()</code> <code>V remove(Object key)</code> <code>V replace(K key,V value)</code> <code>void replaceAll(BiFunction function)</code>
Consultar	<code>V get(Object key)</code> <code>V getOrDefault(Object key, V defaultValue)</code> <code>boolean containsKey(Object key)</code> <code>boolean containsValue(Object value)</code> <code>boolean isEmpty()</code> <code>int size()</code> <code>Set&lt;K&gt; keySet()</code> <code>Collection&lt;V&gt; values()</code> <code>Set&lt;Map.Entry&lt;K,V&gt;&gt; entrySet()</code>
Outros	<code>boolean equals(Object o)</code> <code>int hashCode()</code>



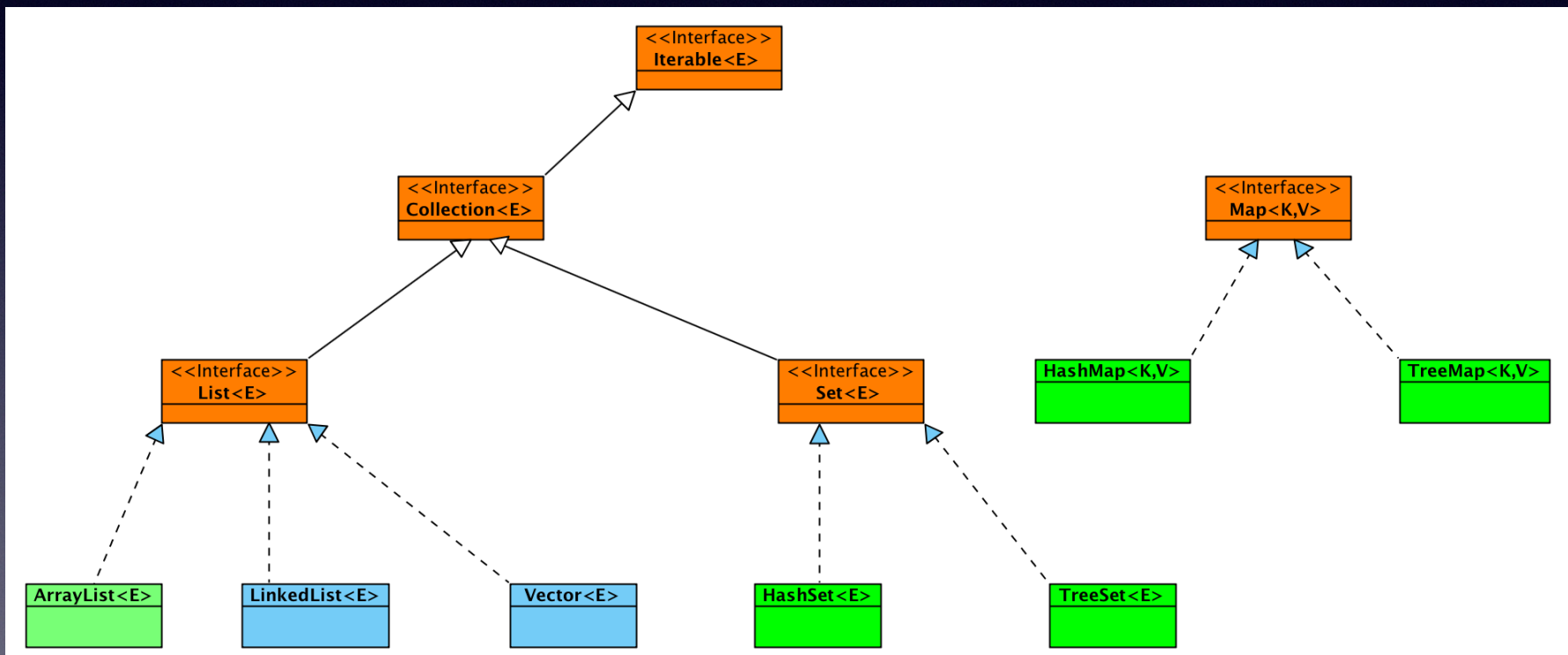
# Coleções associadas a `Map<K,V>`

- **`Set<K> keySet()`**
  - Conjuntos das chaves
- **`Collection<V> values()`**
  - Coleção dos valores
- **`Set<Map.Entry<K,V>> entrySet()`**
  - Conjunto dos pares chave valor

API de  
`Map.Entry<K,V>`

<code>boolean</code>	<b><code>equals(Object o)</code></b> Compares the specified object with this entry for equality.
<code>K</code>	<b><code>getKey()</code></b> Returns the key corresponding to this entry.
<code>V</code>	<b><code>getValue()</code></b> Returns the value corresponding to this entry.
<code>int</code>	<b><code>hashCode()</code></b> Returns the hash code value for this map entry.
<code>V</code>	<b><code>setValue(V value)</code></b> Replaces the value corresponding to this entry with the specified value (optional operation).

# Coleções e Maps





# Regras para utilização de colecções

- Escolher com critério se a colecção a criar deve ser uma lista ou um conjunto (duplicados ou não) ou então uma correspondência entre chaves e valores
- Escolher para sets e maps uma classe de implementação adequada, cf. Hash (sem ordem especial) ou Tree (com comparação pré-definida ou definindo uma ordem de comparação)



# Regras para utilização de colecções

- Nunca usar os métodos pré-definidos **addAll()** ou **putAll()** quando está em causa o encapsulamento. Em vez destes, usar um iterador, interno ou externo, para fazer clone() dos objectos a adicionar
- Sempre que possível, os resultados dos métodos devem ser generalizados para os tipos **List<E>**, **Set<E>** ou **Map<K,V>** em vez de devolverem classes específicas como **ArrayList<E>**, **HashSet<E>**, **TreeSet<E>** ou **HashMap<K,V>**.
  - aumenta-se assim a abstracção



# Ainda sobre ordenações

- Temos visto que podemos ordenar colecções de dados recorrendo:
  - à ordem natural, através do método `compareTo()` (interface `Comparable<T>`)
  - a uma relação de ordem a fornecer, através do método `compare(...)` (interface `Comparator<T>`)



- Temos utilizado o `TreeSet<E>` como mecanismo base para fazer ordenações:
- tirando partido de que o `TreeSet` utiliza uma relação de ordem para colocar os objectos
- como o conjunto não admite repetidos é preciso especial cuidado com comparações que devolvem 0 (são iguais)
- ou se acrescentam mais critérios de comparação ou então alguns elementos são ignorados



- Uma outra alternativa passa por utilizar as `List<E>` para efectuar a ordenação, não tendo que ter de prever a situação de repetição de dados
  - utilizando o método `List.sort(c)`
  - utilizando o `sorted()` e `sorted(c)` das streams

# com recurso a List.sort

this.hoteis é um  
Map<String,Hotel>

```
/**
 * A estratégia de colocar comparators em TreeSet necessita que o método compare
 * , do Comparator, não dê como resultado zero (caso em que o Set não permite ficar
 * com elementos repetidos). Claro que isto nem sempre é possível...
 *
 * Uma forma de permitir continuar a ter repetições é assumir que a estrutura de
 * dados é uma lista e utilizar o método sort (método de classe de Collections)
 * passando como parâmetro um comparator.
 */

public List<Hotel> ordenarHoteisList(Comparator<Hotel> c) {
    List<Hotel> l = new ArrayList<Hotel>();

    l = this.hoteis.values().stream().
        map(Hotel::clone).
        collect(Collectors.toList());

    l.sort(c);
    return l;
}
```



# com recurso a sorted

```
/**
 * Com a utilização de Stream pode efectuar-se directamente sobre a stream a
 * ordenação. Através do método sorted(), para a ordem natural, ou do método
 * sorted(c) para um comparador definido.
 */

public List<Hotel> ordenarHoteisListStream(Comparator<Hotel> c) {
    return this.hoteis.values().stream()
        .map(Hotel::clone).sorted(c).collect(Collectors.toList());
}
```

# Mais sobre Collectors

```
static <T> Collector<T,?,List<T>>
```

```
toList()
```

Returns a Collector that accumulates the input elements into a new List.

```
static <T> Collector<T,?,Set<T>>
```

```
toSet()
```

Returns a Collector that accumulates the input elements into a new Set.

```
static <T,C extends Collection<T>>  
Collector<T,?,C>
```

```
toCollection(Supplier<C> collectionFactory)
```

Returns a Collector that accumulates the input elements into a new Collection, in encounter order.

```
static <T,K,U> Collector<T,?,Map<K,U>>
```

```
toMap(Function<? super T,? extends K> keyMapper,  
Function<? super T,? extends U> valueMapper)
```

Returns a Collector that accumulates elements into a Map whose keys and values are the result of applying the provided mapping functions to the input elements.

```
static <T,K,U,M extends Map<K,U>>  
Collector<T,?,M>
```

```
toMap(Function<? super T,? extends K> keyMapper,  
Function<? super T,? extends U> valueMapper,  
BinaryOperator<U> mergeFunction,  
Supplier<M> mapSupplier)
```

Returns a Collector that accumulates elements into a Map whose keys and values are the result of applying the provided mapping functions to the input elements.



# Mais sobre *reduce*

reduce  
pré-definido

```
double sum = alunos.stream().mapToDouble(Aluno::getNota).sum();
```

**Optional<T> reduce(BinaryOperator<T> accumulator)**

Performs a **reduction** on the elements of this stream, using an **associative** accumulation function, and returns an **Optional** describing the reduced value, if any.

**T reduce(T identity, BinaryOperator<T> accumulator)**

Performs a **reduction** on the elements of this stream, using the provided identity value and an **associative** accumulation function, and returns the reduced value.

**<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)**

Performs a **reduction** on the elements of this stream, using the provided identity, accumulation and combining functions.

```
OptionalDouble sum = alunos.stream().mapToDouble(Aluno::getNota).reduce((ac, v) -> ac+v);
```

```
double sum = alunos.stream().mapToDouble(Aluno::getNota).reduce(0.0, (ac, v) -> ac+v);
```

```
double sum = alunos.stream().reduce(0.0,  
                                   (ac, al) -> ac+al.getNota(),  
                                   (ac1, ac2) -> ac1+ac2);
```

# Mais sobre Optional

- **Optional<T>**
  - **OptionalDouble**
  - **OptionalInt**
  - **OptionalLong**
- Alguns métodos relevantes...

T	<b>get()</b> If a value is present in this Optional, returns the value, otherwise throws NoSuchElementException.
boolean	<b>isPresent()</b> Return true if there is a value present, otherwise false.
T	<b>orElseGet(Supplier&lt;? extends T&gt; other)</b> Return the value if present, otherwise invoke other and return the result of that invocation.
T	<b>orElse(T other)</b> Return the value if present, otherwise return other.



# Exemplo

```
/**
 * Média da turma
 *
 * @return um double com a média da turma (zero se turma vazia)
 */
public double media() {
    OptionalDouble res = alunos.stream()
                               .mapToDouble(Aluno::getNota)
                               .average();

    return res.orElse(0.0);
}
```

Simplificando...

```
/**
 * Média da turma
 *
 * @return um double com a média da turma (zero se turma vazia)
 */
public double media() {
    return alunos.stream()
                 .mapToDouble(Aluno::getNota)
                 .average()
                 .orElse(0.0);
}
```

# Mais sobre mutable reductions

`<R,A> R collect(Collector<? super T,A,R> collector)`

Performs a **mutable reduction** operation on the elements of this stream using a Collector.

`<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)`

Performs a **mutable reduction** operation on the elements of this stream.

```
public Set<String> getNomes() {  
    return alunos.stream()  
        .map(Aluno::getNome)  
        .collect(Collectors.toSet());  
}
```

Não temos garantia sobre o tipo de Set

```
public Set<Aluno> getAlunos() {  
    return alunos.stream()  
        .map(Aluno::clone)  
        .collect(Collectors.toCollection(TreeSet::new));  
}
```

Podemos indicar que tipo de colecção pretendemos

```
public Set<String> getNomes() {  
    return alunos.stream()  
        .collect(HashSet::new, (s, al) -> s.add(al.getNome()), HashSet::addAll);  
}
```

Fazer a transformação durante o collect...  
(desnecessariamente complicado, neste caso)



# Exemplo - criar um Map

```
public Map<Integer,Aluno> getAlunos() {  
    Map<Integer, Aluno> copia = new HashMap<>();  
  
    for (Map.Entry<Integer,Aluno> e: alunos.entrySet())  
        copia.put(e.getKey(), e.getValue().clone());  
  
    return copia;  
}
```

VS

```
public Map<Integer,Aluno> getAlunos() {  
    return alunos.entrySet()  
        .stream()  
        .collect(Collectors.toMap((e)->e.getKey(),  
                                   (e)->e.getValue().clone()));  
}
```

# ... ainda sobre Collectors

```
static <T,K> Collector<T,?,Map<K,List<T>>>
```

```
groupBy(Function<? super T,? extends  
K> classifier)
```

Returns a Collector implementing a "group by" operation on input elements of type T, grouping elements according to a classification function, and returning the results in a Map.

```
static <T,K,A,D> Collector<T,?,Map<K,D>>
```

```
groupBy(Function<? super T,? extends  
K> classifier, Collector<? super T,A,D> downstream)
```

Returns a Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector.

```
static <T,K,D,A,M extends Map<K,D>>  
Collector<T,?,M>
```

```
groupBy(Function<? super T,? extends  
K> classifier, Supplier<M> mapFactory, Collector<?  
super T,A,D> downstream)
```

Returns a Collector implementing a cascaded "group by" operation on input elements of type T, grouping elements according to a classification function, and then performing a reduction operation on the values associated with a given key using the specified downstream Collector.



# Exemplos

```
/**
 * Calcular um Map de nota para lista de alunos com essa nota.
 */
public Map<Double, List<Aluno>> porNota() {
    Map<Double, List<Aluno>> res = new TreeMap<>();

    for(Aluno a : alunos.values()) {
        double nota = a.getNota();
        if (!res.containsKey(nota))
            res.put(nota, new ArrayList<>());
        res.get(nota).add(a.clone());
    }
    return res;
}
```

Percorrer o Map  
alunos (com values())

A chave do Map é a  
nota.

... se ainda  
não existe aquela  
entrada cria-se o par  
chave/lista vazia

VS

```
/**
 * Calcular um Map de nota para lista de alunos com essa nota.
 */
public Map<Double, List<Aluno>> porNota() {
    return alunos.values()
        .stream()
        .collect(Collectors.groupingBy(Aluno::getNota,
            Collectors.mapping(Aluno::clone, Collectors.toList())));
}
```

A chave do Map  
é a nota.

Vamos  
guardar cópias de  
cada aluno...

...num List.

# Exemplos

## **import static**

`java.util.stream.Collectors.*;`

A utilização de `import static` permite simplificar as expressões, eliminando a necessidade de 'Collectors'.

```
/**
 * Calcular um TreeMap de nota para Set de nomes dos alunos com essa nota.
 */
public TreeMap<Double, Set<String>> nomesPorNota() {

    return alunos.values()
        .stream()
        .collect(groupingBy(Aluno::getNota,
                           TreeMap::new,
                           mapping(Aluno::getNome, toSet())));
}
```

A chave do Map é a nota

Queremos um TreeMap

Queremos guardar o nome dos alunos

Os nomes vão ser guardados num Set



# Side effects

- Em geral, devem evitar-se efeitos laterais nos métodos/expressões lambda utilizados nas Streams
  - vão complicar paralelização das streams no futuro
- **forEach()** (e **peek()**) operam via efeitos laterais pelo que devem ser utilizados com cuidado
- Em muitos casos, os efeitos laterais podem ser evitados...

```
ArrayList<String> results = new ArrayList<>();  
stream.filter(s -> pattern.matcher(s).matches())  
    .forEach(s -> results.add(s)); // Unnecessary use of side-effects!
```

```
List<String>results =  
    stream.filter(s -> pattern.matcher(s).matches())  
        .collect(Collectors.toList()); // No side-effects!
```