

Iteradores internos (a partir de Java 8)

Todas as colecções implementam o método: **forEach()**

Aceita uma função para *trabalhar* em todos os elementos da coleção

É implementado com um foreach...

```
default void forEach(Consumer<? super T> action) {  
    Objects.requireNonNull(action);  
    for (T t : this) {  
        action.accept(t);  
    }  
}
```

Iterador externo

```
/**
 * Subir a nota a todos os alunos
 *
 * @param bonus int valor a subir.
 */
public void aguaBenta(int bonus) {
    for(Aluno a: lstAlunos)
        a.sobeNota(bonus);
}
```

Iterador interno - forEach()

```
/**
 * Subir a nota a todos os alunos
 *
 * @param bonus int valor a subir.
 */
public void aguaBenta(int bonus) {
    lstAlunos.forEach((Aluno a) -> {a.sobeNota(bonus);});
}
```

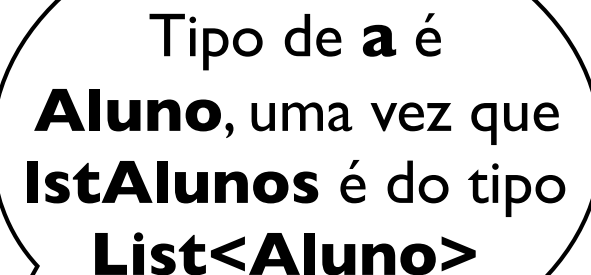
Expressões Lambda

(Tipo p, ...) -> {corpo do método}

Um método *anônimo*, que pode ser passado como parâmetro

Expressão pode ser simplificada:

```
/**
 * Subir a nota a todos os alunos
 *
 * @param bonus int valor a subir.
 */
public void aguaBenta(int bonus) {
    lstAlunos.forEach(a -> a.sobeNota(bonus));
}
```

A speech bubble with a black outline and a tail pointing towards the bottom left, containing text about the type of variable 'a' in the lambda expression.

Tipo de **a** é **Aluno**, uma vez que **lstAlunos** é do tipo **List<Aluno>**

Streams

Todas as colecções implementam o método **stream()**

Streams: sequências de valores que podem ser passados numa *pipeline* de operações.

As operações alteram os valores (produzindo novas Streams ou *reduzindo* o valor a um só)

```
public int quantosPassam() {  
    int qt = 0;  
  
    for(Aluno a: lstAlunos)  
        if (a.passa()) qt++;  
  
    return qt;  
}
```

```
public long quantosPassam() {  
  
    return lstAlunos.stream().filter(a -> a.passa()).count();  
}
```

Colecções implementam método **stream()**

Produz uma Stream

Alguns dos principais métodos da API de **Stream**

`allMatch()` - determina se todos os elementos fazem match com o predicado fornecido

`anyMatch()` - determina se algum elemento faz match

`noneMatch()` - determina se nenhum elemento faz match

`count()` - conta os elementos da Stream

`filter()` - filtra os elementos da Stream usando um predicado

`map()` - transforma os elementos da Stream usando uma função

`collect()` - junta os elementos da Stream numa lista ou String

`reduce()` - realiza uma redução (fold)

`sorted()` - ordena os elementos da Stream

`toArray()` - retorna um array com os elementos da Stream

alguemPassa() - utilizando Streams...

```
/**
 * Alguns alunos passam?
 *
 * @return true se algum aluno passa
 */
public boolean alguemPassa() {
    return lstAlunos.stream().anyMatch(a -> a.passa());
}
```

```
/**
 * Alguns alunos passam?
 *
 * @return true se algum aluno passa
 */
public boolean alguemPassa() {
    boolean alguem = false;
    Iterator<Aluno> it = lstAlunos.iterator();
    Aluno a;

    while(it.hasNext() && !alguem) {
        a = it.next();
        if (a.passa())
            alguem = true;
    }
    return alguem;
}
```

Referências a métodos

Classe::método

Permitem referir um método pelo seu nome

Úteis nas expressões lambda

Objecto que recebe a mensagem está implícito no contexto

```
public boolean alguemPassa() {  
    return lstAlunos.stream().anyMatch(Aluno::passa);  
}
```

getLstAlunos()

```
public List<Aluno> getLstAlunos() {  
    return lstAlunos.stream().map(Aluno::clone).collect(Collectors.toList());  
}
```

```
public List<Aluno> getLstAlunos() {  
    List<Aluno> res = new ArrayList<>();  
  
    for(Aluno a: lstAlunos)  
        res.add(a.clone());  
    return res;  
}
```


remover alunos utilizando Streams

```
/**
 * Remover notas mais baixas
 *
 * @param nota a nota limite
 */
public void removerPorNota(int nota) {
    lstAlunos = lstAlunos.stream()
        .filter(a -> a.getNota() >= nota)
        .collect(Collectors.toList());
}
```

mas...

```
public void removerPorNota(int nota) {
    lstAlunos.removeIf(a -> a.getNota() < nota);
}
```

```
/**
 * Remover notas mais baixas
 *
 * @param nota a nota limite
 */
public void removerPorNota(int nota) {
    Iterator<Aluno> it = lstAlunos.iterator();
    Aluno a;

    while(it.hasNext()) {
        a = it.next();
        if (a.getNota() < nota)
            it.remove();
    }
}
```

Existem Steams Especificas para os tipos primitivos

IntStream - **mapToInt(...)**

DoubleStream - **mapToDouble(...)**

...

Alguns dos principais métodos específicos

average() - determina a média

max() - determina o máximo

min() - determina o mínimo

sum() - determina a soma

media() - utilizando Streams...

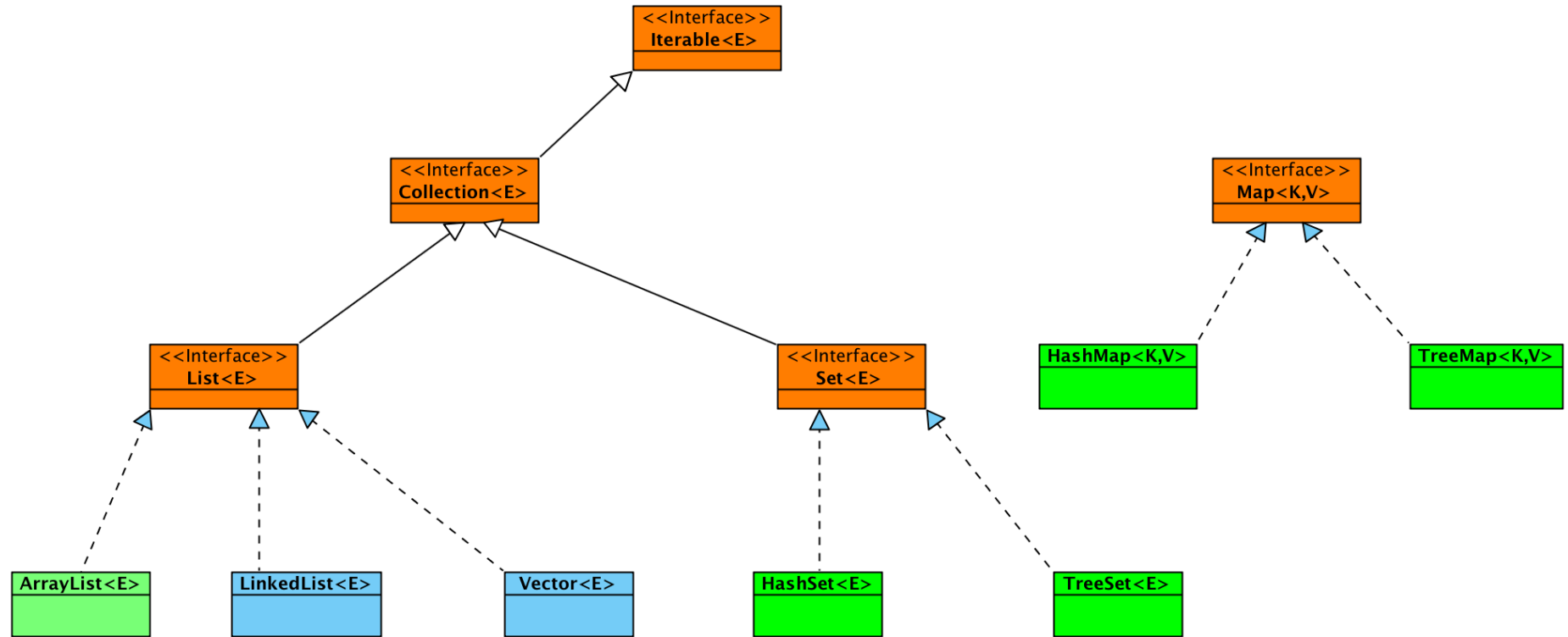
```
/**
 * Média da turma
 *
 * @return um double com a média da turma
 */
public double media() {
    double tot = lstAlunos.stream()
                           .mapToDouble(Aluno::getNota)
                           .sum();
    return tot/lstAlunos.size();
}
```

```
public double media() {
    double tot = 0.0;

    for(Aluno a: lstAlunos)
        tot += a.getNota();

    return tot/lstAlunos.size();
}
```

Coleções e Maps



Set<E>

Adicionar elementos	<code>boolean add(E e)</code> <code>boolean addAll(Collection c)</code>
Alterar o Set	<code>void clear()</code> <code>boolean remove(Object o)</code> <code>boolean removeAll(Collection c)</code> <code>boolean retainAll(Collection c)</code> <code>boolean removeIf(Predicate p)</code>
Consultar	<code>boolean contains(Object o)</code> <code>boolean containsAll(Collection c)</code> <code>boolean isEmpty()</code> <code>int size()</code>
Iteradores externos	<code>Iterator<E> iterator()</code>
Iteradores internos	<code>Stream<E> stream()</code> <code>void forEach(Consumer c)</code>
Outros	<code>boolean equals(Object o)</code> <code>int hashCode()</code>

Set<E>

Utilizar sempre que se quer garantir ausência de elementos repetidos

O método add testa se o objecto existe

O método contains utiliza a lógica do equals, mas não só...

Duas implementações: **HashSet<E>** e **TreeSet<E>**

HashSet<E>

Utiliza uma tabela de Hash para guardar os elementos.

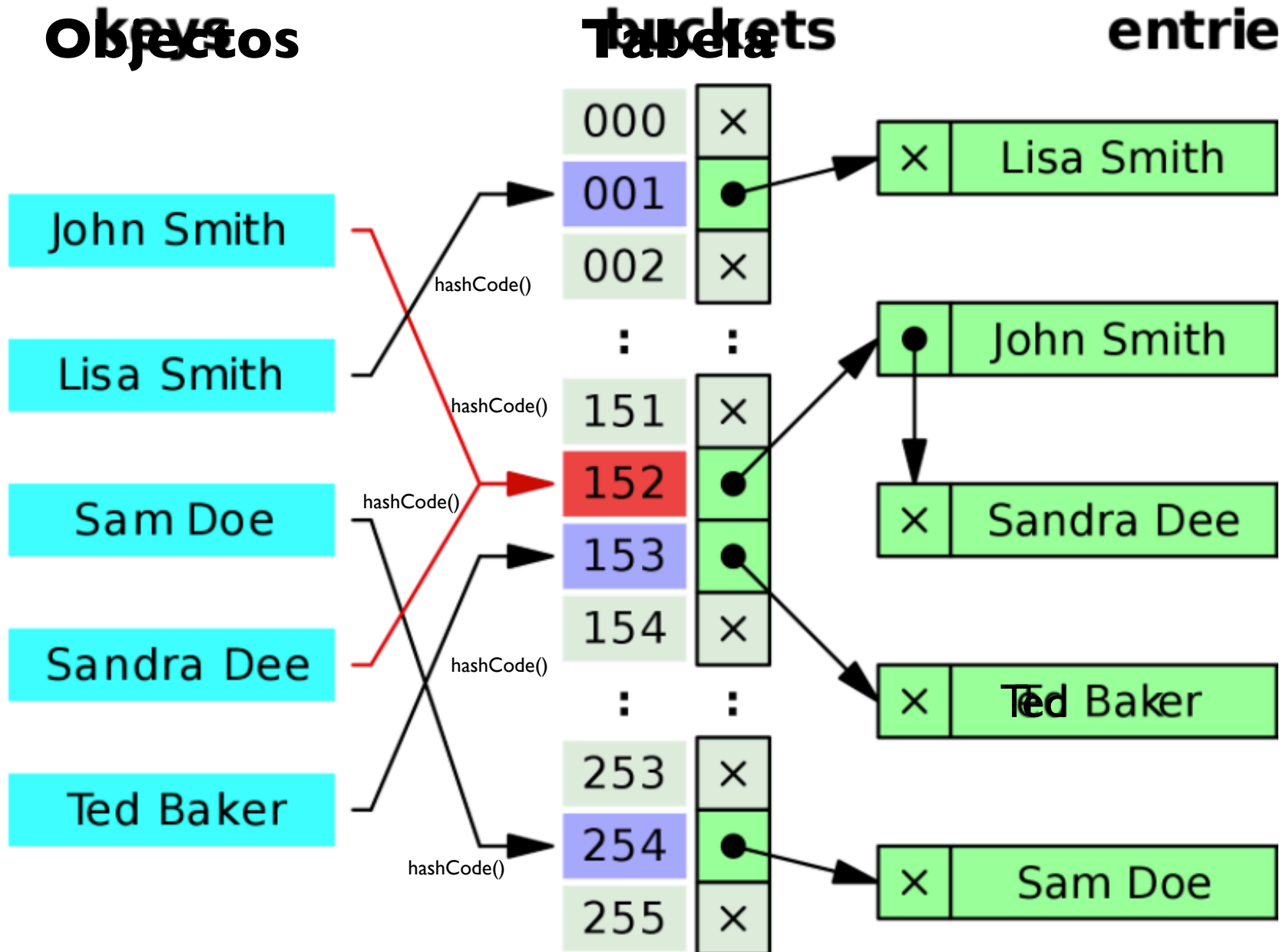
O método **add** calcula o valor de hash do objecto a adicionar para determinar a sua posição na estrutura de dados

O método **contains** necessita de saber o valor de hash do objecto para determinar a posição em que o encontra

Logo, não chega ter o **equals** definido

é necessário ter o método **hashCode()**

Tabelas de hash



Método hashCode()

Sempre que se define o método **equals**, deve definir-se também o método **hashCode()**

objectos iguais devem ter o mesmo código de hash

Se **hashCode()** não for definido é utilizada a implementação por omissão, logo:

recorre à referência do objecto

objectos iguais (em valor) podem ter códigos diferentes!

Método hashCode()

Exemplo

nome é String

número é int

nota é double

```
public int hashCode() {  
    int hash = 7;  
    long aux;  
  
    hash = 31*hash + nome.hashCode();  
    hash = 31*hash + numero;  
    aux = Double.doubleToLongBits(nota);  
    hash = 31*hash + (int)(aux^(aux >>> 32));  
    return hash;  
}
```

Implementar o hashCode()

(exemplo!)

Definir **int hash = x ;** //(x diferente de 0)

Calcular o código de hash de cada var. instância **v** conforme o seu tipo:

boolean: **(v ? 0 : 1);**

byte, char, short ou int: **(int)v;**

long: **(int)(v ^ (v >>> 32));**

float: **Float.floatToIntBits(v);**

double: calcular **Double.doubleToLongBits(v)** e usar a regra dos long no resultado

objectos: **v.hashCode()**, ou **0** se **v == null**;

arrays: tratar cada elemento do array como uma variável de inst.

Combinar cada um dos valores calculados acima no resultado do seguinte modo: **hash = 37 * hash + valor;**

return result;

TreeSet<E>

Utiliza uma árvore binária auto-balanceada do tipo *Red-Black* para guardar os elementos.

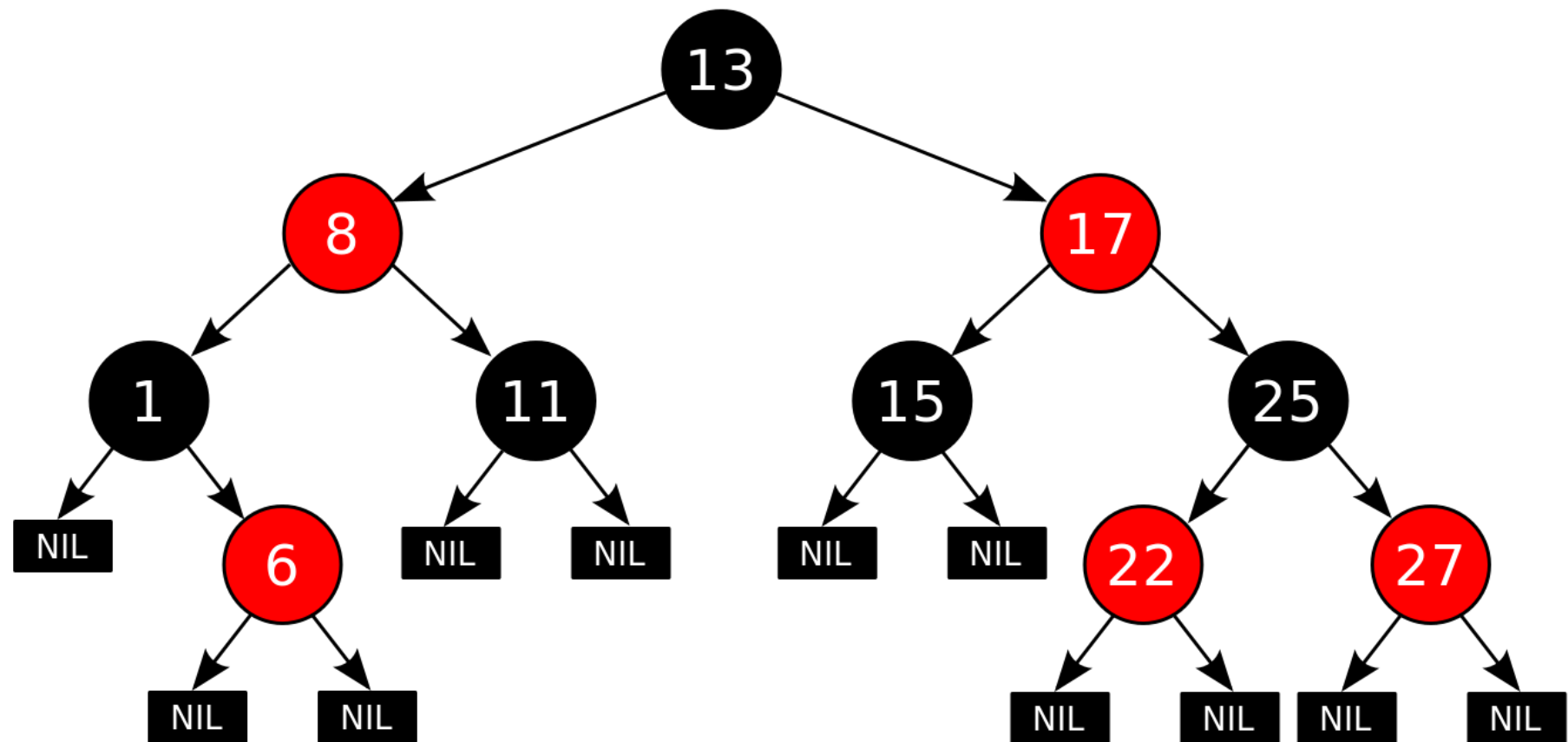
É necessário fornecer um método de comparação dos objectos

compareTo() - na classe **E**

compare() - num **Comparator**

sem este método de comparação não é possível utilizar o TreeSet, a não ser para tipos de dados simples (String, Integer, etc.)

Red-black self-balancing binary search tree



Método compareTo()

Define a ordem “natural” das instâncias de uma dada classe

Definido como um método de instância

Compara o objecto receptor com outro passado como parâmetro

Se objectos são iguais

resultado: **0**

Se objecto receptor é “maior”

resultado > 0 (neste caso **1**)

Se objecto receptor é “menor”

resultado < 0 (neste caso **-1**)

```
public int compareTo(Aluno a) {  
    int numA = a.getNumero();  
    int res;  
  
    if (this.numero==numA)  
        res = 0;  
    else if (numero>numA)  
        res = 1;  
    else  
        res = -1;  
    return res;  
}
```

Método compareTo()

Classe deve implementar **Comparable<T>**

public class Aluno implements Comparable<Aluno>

Ordem natural com base no número (versão alternativa)

```
public int compareTo(Aluno a) {  
    if (this.numero==a.getNumero())  
        return 0;  
    if (this.numero>a.getNumero())  
        return 1;  
    return 0;  
}
```

Ordem natural com base no nome

```
public int compareTo(Aluno a) {  
    return this.nome.compareTo(a.getNome());  
}
```

No entanto, só pode existir uma ordem natural (um método **compareTo()**) em cada classe.

TreeSet<E>

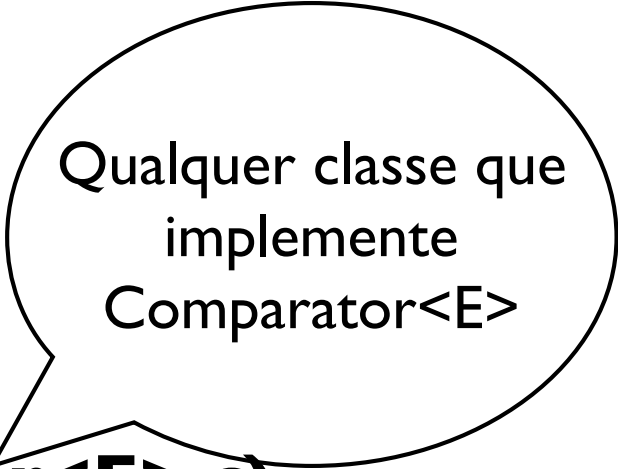
Construtores

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 de Aluno com ordenação por comparador

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

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

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

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.

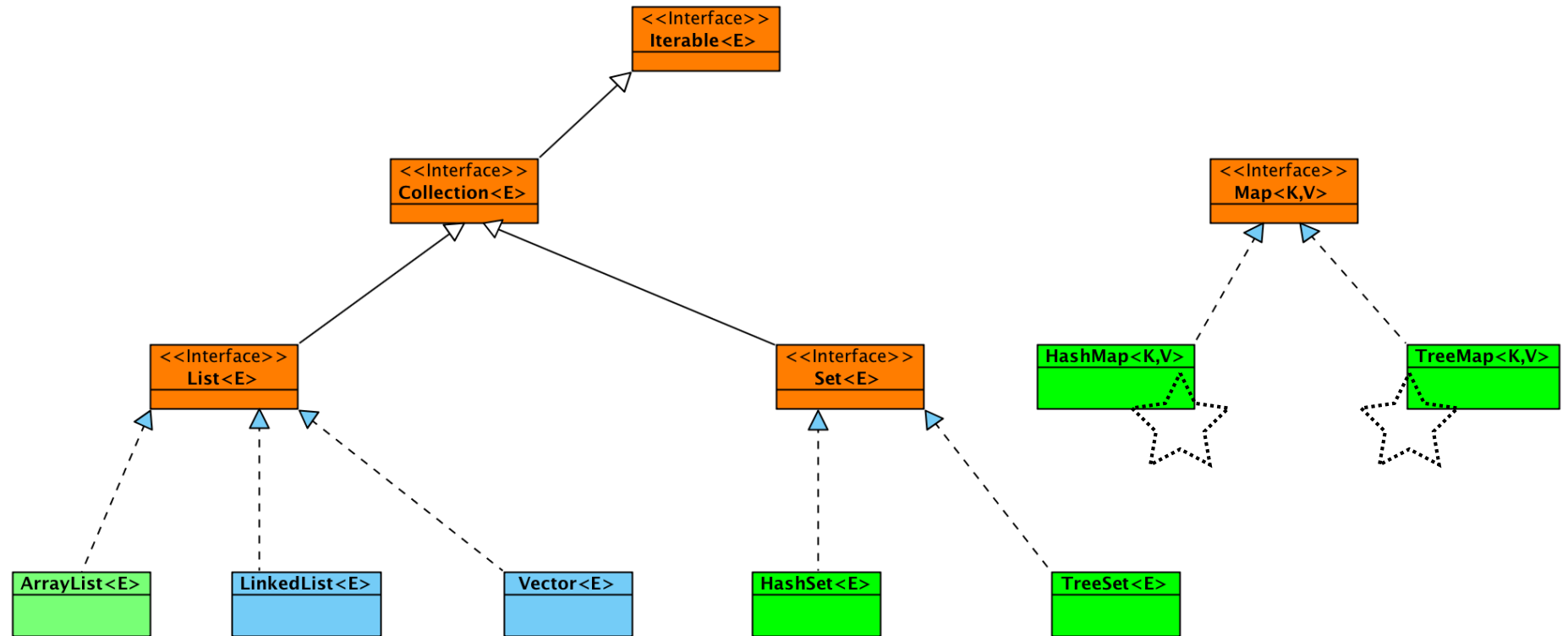
Uma outra forma é recorrer a um método anónimo, escrito sob a forma de uma expressão lambda.

```
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>

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<V> keySet()</code> <code>Collection<V> values()</code> <code>Set<Map.Entry<K,V>> entrySet()</code>
Outros	<code>boolean equals(Object o)</code> <code>int hashCode()</code>

Coleções associadas a **Map<K,V>**

Set<V> keySet()

Conjuntos das chaves

Collection<V> values()

Colecção dos valores

Set<Map.Entry<K,V>> entrySet()

Conjunto
dos pares
chave valor

boolean equals(Object o)

Compares the specified object with this entry for equality.

K getKey()

Returns the key corresponding to this entry.

V getValue()

Returns the value corresponding to this entry.

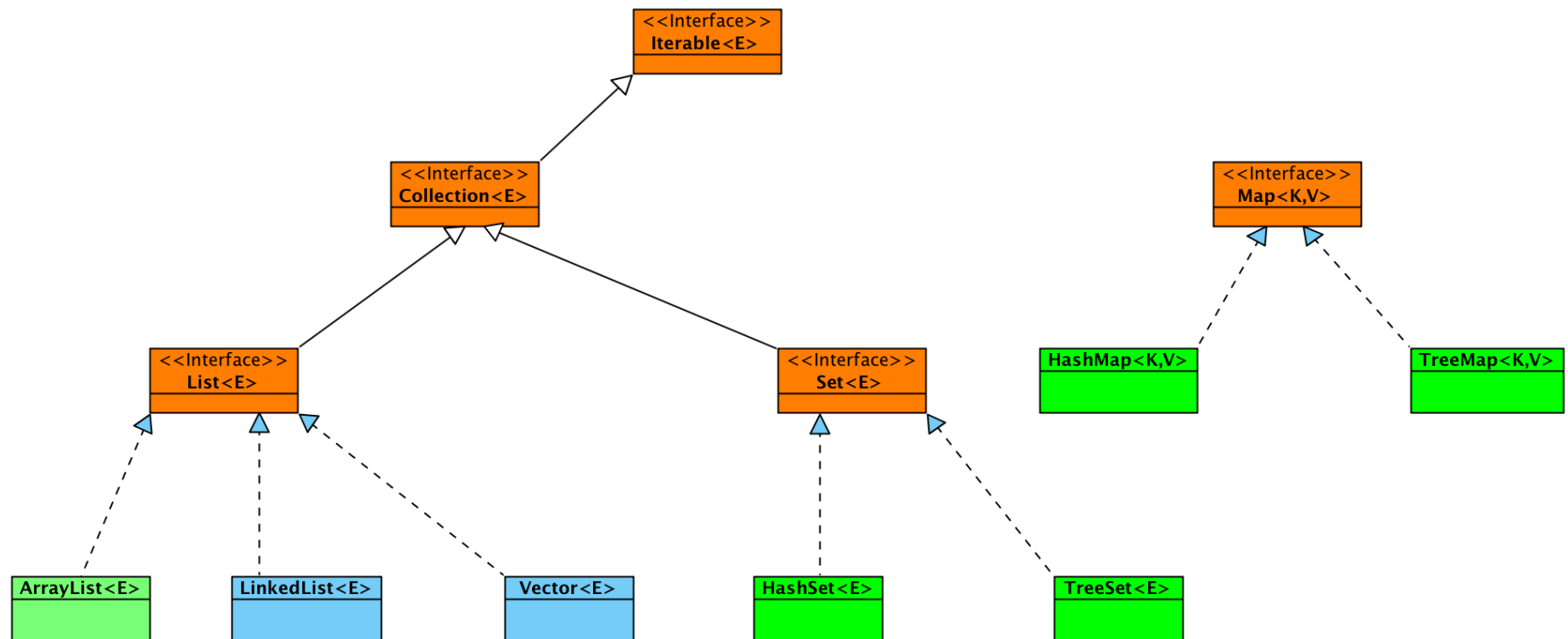
int hashCode()

Returns the hash code value for this map entry.

V setValue(V value)

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 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

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* (aka *fold*)

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

`get()`

If a value is present in this `Optional`, returns the value, otherwise throws `NoSuchElementException`.

boolean

`isPresent()`

Return true if there is a value present, otherwise false.

T

`orElseGet(Supplier<? extends T> other)`

Return the value if present, otherwise invoke `other` and return the result of that invocation.

T

`orElse(T other)`

Return the value if present, otherwise return `other`.