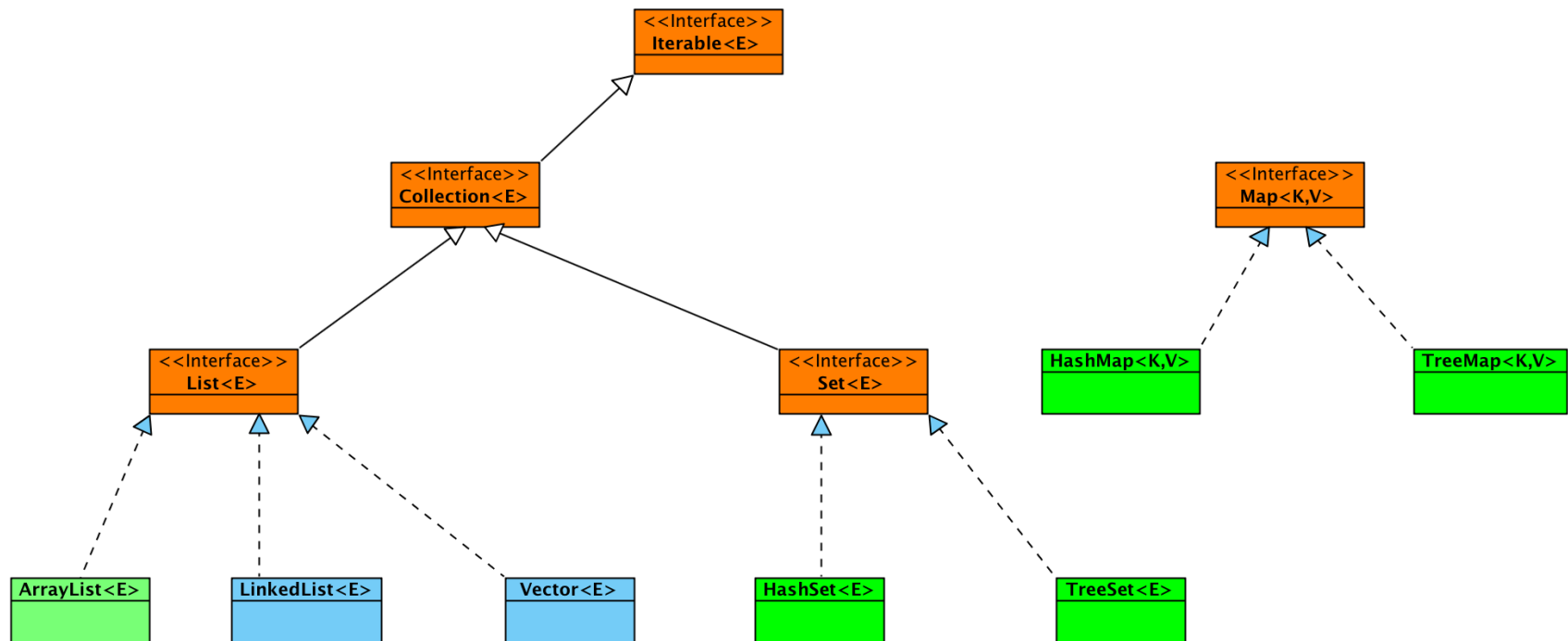
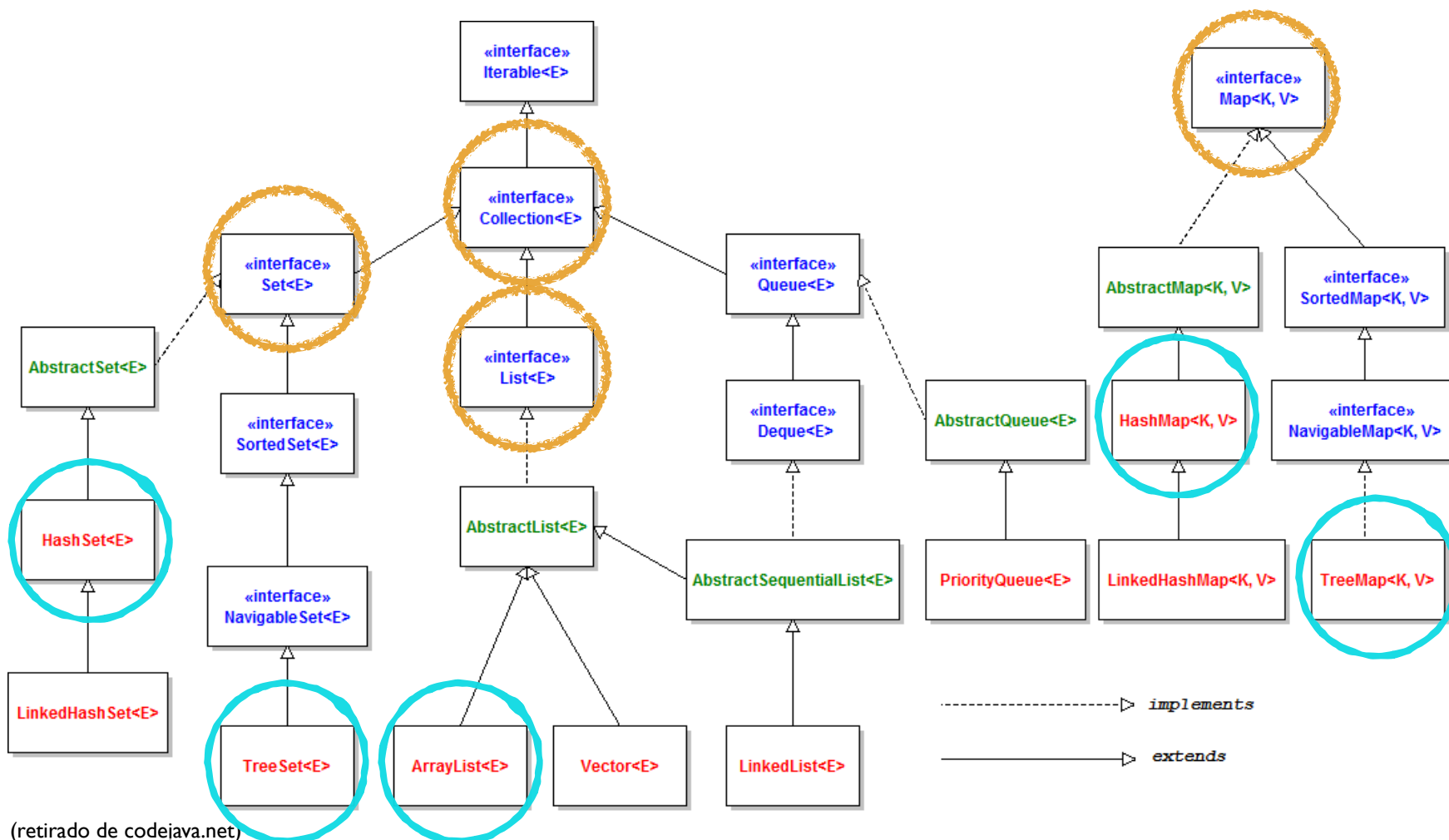


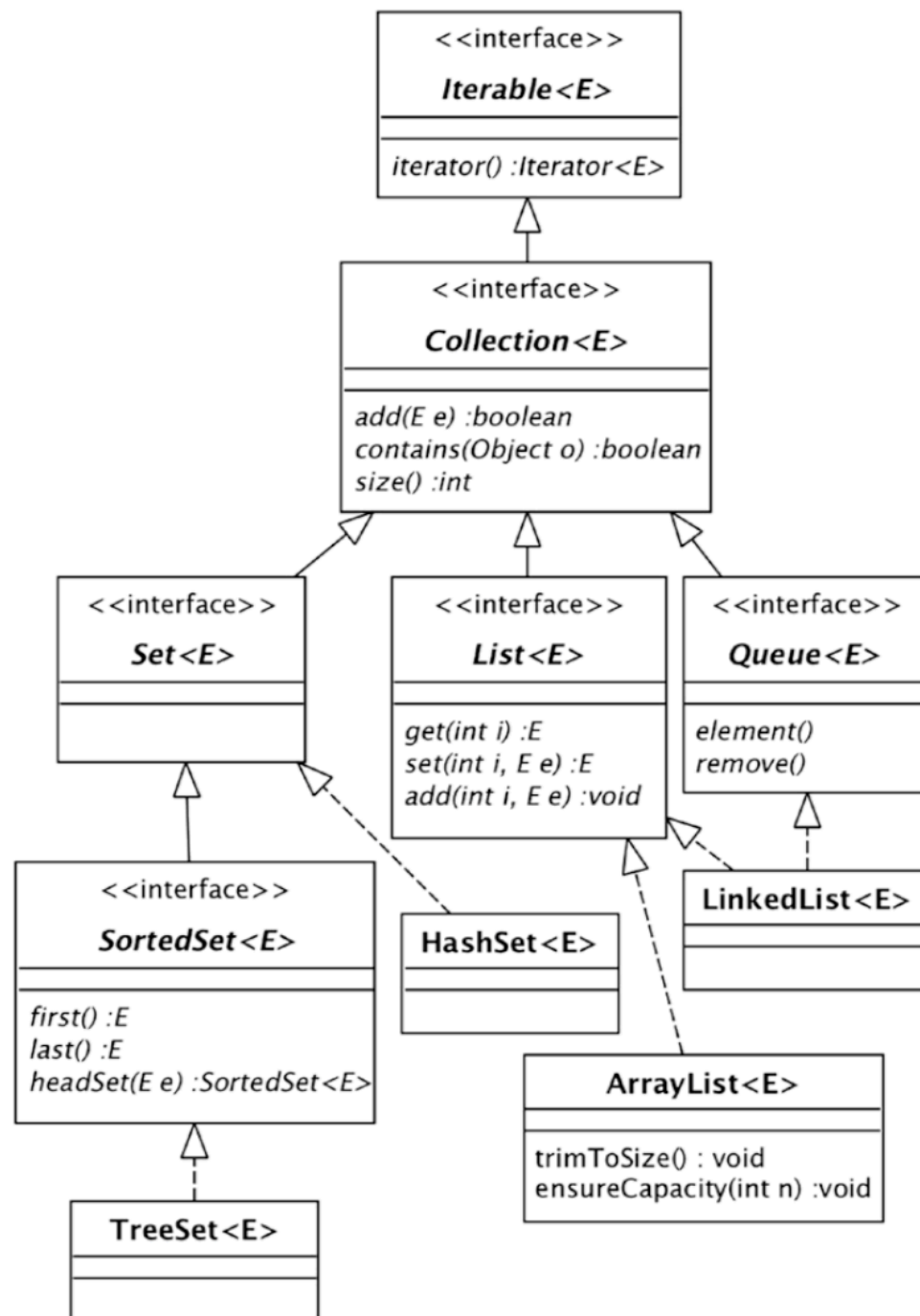
- Para cada API (interface) existem diversas implementações (a escolher consoante critérios do programador)
- *Vamos estudar as que estão a verde*



Interfaces e Implementações do JFC



(retirado de codejava.net)



- todas as coleções permitem ser iteradas
- implementações de List<E> adicionam métodos com indicação de posição

(retirado de Java Program Design, Edward Sciore)

ArrayList<E>

- As classes da Java Collections Framework são exemplos muito interessantes de codificação
- Como o código destas classes está escrito em Java é possível ao programador observar como é que foram implementadas

ArrayList<E>: v.i. e construtores

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    private static final long serialVersionUID = 8683452581122892189L;

    /**
     * The array buffer into which the elements of the ArrayList are stored.
     * The capacity of the ArrayList is the length of this array buffer.
     */
    private transient Object[] elementData;

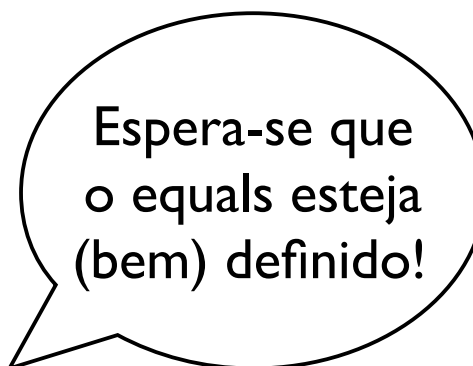
    /**
     * The size of the ArrayList (the number of elements it contains).
     *
     * @serial
     */
    private int size;

    /**
     * Constructs an empty list with the specified initial capacity.
     *
     * @param initialCapacity the initial capacity of the list
     * @throws IllegalArgumentException if the specified initial capacity
     *         is negative
     */
    public ArrayList(int initialCapacity) {
        ...
        this.elementData = new Object[initialCapacity];
    }

    /**
     * Constructs an empty list with an initial capacity of ten.
     */
    public ArrayList() {
        this(10);
    }
}
```

ArrayList<E>: existe?

```
public boolean contains(Object o) {  
    return indexOf(o) >= 0;  
}  
  
public int indexOf(Object o) {  
    if (o == null) {  
        for (int i = 0; i < size; i++)  
            if (elementData[i]==null)  
                return i;  
    } else {  
        for (int i = 0; i < size; i++)  
            if (o.equals(elementData[i]))  
                return i;  
    }  
    return -1;  
}
```



Espera-se que
o equals esteja
(bem) definido!

ArrayList<E>: inserir

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1); // Increments modCount!!
    elementData[size++] = e;
    return true;
}

public void add(int index, E element) {
    rangeCheckForAdd(index);

    ensureCapacityInternal(size + 1); // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1,
        size - index);
    elementData[index] = element;
    size++;
}
```

ArrayList<E>: get e set

```
public E get(int index) {  
    rangeCheck(index);  
  
    return elementData(index);  
}  
  
public E set(int index, E element) {  
    rangeCheck(index);  
  
    E oldValue = elementData(index);  
    elementData[index] = element;  
    return oldValue;  
}
```


Colecções Java

- Tipos de colecções disponíveis:
 - listas (definição em `List<E>`)
 - conjuntos (definição em `Set<E>`)
 - queues (definição em `Queue<E>`)
- noção de família (muito evidente) nas APIs de cada um destes tipos de colecções.

List<E>

- Utilizar sempre que precise de manter ordem
- O método `add` não testa se o objecto existe na colecção (admite repetições)
- O método `contains` verifica sempre o resultado de `equals`
- Implementação utilizada: **`ArrayList<E>`**

ArrayList<E>

Construtores	ArrayList<E>(); ArrayList<E>(int initialCapacity)
Adicionar elementos	boolean add(E e); void add(int index, E element); boolean addAll(Collection c); boolean addAll(int index, Collection c)
Remover elementos	boolean remove(Object o); E remove(int index); boolean removeAll(Collection c) boolean retainAll(Collection c) boolean removeIf(Predicate p)
Consultar	E get(int index); int indexOf(Object o); int lastIndexOf(Object o); boolean contains(Object o) boolean containsAll(Collection c) boolean isEmpty(); int size()
Alterar elementos	E set(int index, E element); void clear();
Iteradores externos	Iterator<E> iterator()
Iteradores internos	Stream<E> stream(); void forEach(Consumer c)
Outros	boolean equals(Object o); T[] toArray(T[] a)

```

import java.util.ArrayList;
public class TesteArrayList {
    public static void main(String[] args) {

        Circulo c1 = new Circulo(2,4,4.5);
        Circulo c2 = new Circulo(1,4,1.5);
        Circulo c3 = new Circulo(2,7,2.0);
        Circulo c4 = new Circulo(3,3,2.0);
        Circulo c5 = new Circulo(2,6,7.5);

        ArrayList<Circulo> circs = new ArrayList<Circulo>();
        circs.add(c1.clone());
        circs.add(c2.clone());
        circs.add(c3.clone());

        System.out.println("Num elementos = " + circs.size());
        System.out.println("Posição do c2 = " + circs.indexOf(c2));

        for(Circulo c: circs)
            System.out.println(c.toString());
    }
}

```

estratégia de composição

Percorrer uma coleção

- Podemos utilizar o ciclo for(each) para percorrer uma coleção:

```
/**
 * Média da turma
 *
 * @return um double com a média da turma
 */
public double media() {
    double tot = 0.0;

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

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

```
/**
 * Quantos alunos passam?
 *
 * @return um int com nº alunos que passa
 */
public int quantosPassam() {
    int qt = 0;

    for(Aluno a: lstAlunos)
        if (a.passa()) qt++;

    return qt;
}
```

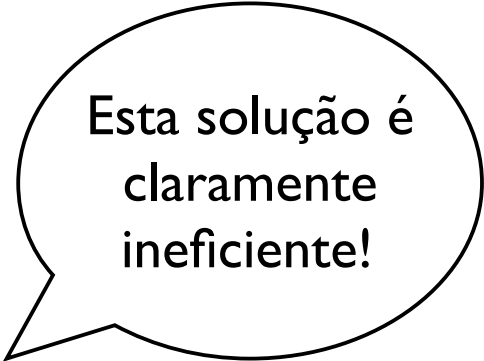
```
public boolean passa() {
    return this.nota >= Aluno.NOTA_PARA_PASSAR;
}
```

Na classe **Aluno**

- ... mas...
- podemos querer parar antes do fim
- podemos não ter acesso à posição do elemento na colecção (no caso dos conjuntos)
- estamos sempre a repetir o código do ciclo

```
/**
 * Algum aluno passa?
 *
 * @return true se algum aluno passa
 */
public boolean alguemPassa() {
    boolean alguem = false;

    for(Aluno a: lstAlunos)
        if (a.passa())
            alguem = true;
    return alguem;
}
```



Esta solução é
claramente
ineficiente!

- logo, é necessário um mecanismo mais flexível para percorrer colecções

Iteradores externos

- O **Iterator** é um padrão de concepção bem conhecido e que permite providenciar uma forma de aceder aos elementos de uma colecção de objectos, sem que seja necessário saber qual a sua representação interna
- basta para tal, que todas as colecções saibam criar um iterator!
- não precisamos saber como tal é feito!

- Um iterador de uma lista poderia ser:



- o iterador precisa de ter mecanismos para:
 - aceder ao objecto apontado
 - avançar
 - determinar se chegou ao fim

- Iterator API

Method Summary

Methods

Modifier and Type	Method and Description
boolean	hasNext() Returns <code>true</code> if the iteration has more elements.
E	next() Returns the next element in the iteration.
void	remove() Removes from the underlying collection the last element returned by this iterator (optional operation).

- Utilizando Iterators...

```
/**
 * Algum aluno passa?
 *
 * @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();
        alguem = a.passa();
    }
    return alguem;
}
```

- remover alunos...

```
/**
 * 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();
    }
}
```

Iterator<E>

- Em resumo...
- Todas as colecções implementam o método: **Iterator<E> iterator()** que cria um iterador activo sobre a colecção
- Padrão de utilização:

```
Iterator<E> it = colecção.iterator();  
E elem;
```

```
while(it.hasNext()) {  
    elem = it.next();  
    // fazer algo com elem  
}
```

- Procurar:

```
boolean encontrado = false;  
Iterator<E> it = coleção.iterator();  
E elem;  
  
while(it.hasNext() && !encontrado) {  
    elem = it.next();  
    if (criterio de procura sobre elem)  
        encontrado = true;  
}  
// fazer alguma coisa com elem ou com encontrado
```

- Remover:

```
Iterator<E> it = coleção.iterator();  
E elem;  
  
while(it.hasNext()) {  
    elem = it.next();  
    if (criterio sobre elem)  
        it.remove();  
}
```

Iteradores internos

- Todas as colecções implementam o método: **forEach()**
- Aceita uma função para *trabalhar* em todos os elementos da coleção
- É implementado com um for each...

```
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);});
}
```

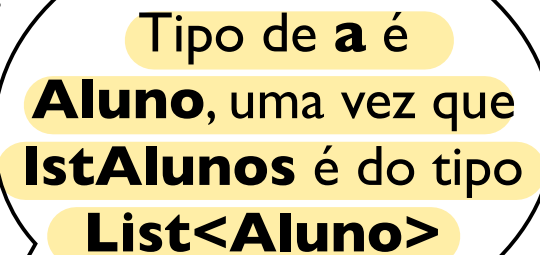
Expressões Lambda

- **(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));
}
```



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

Streams

- Todas as coleções 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();  
}
```

→ todos que respeitam o filtro vão
para um site provisório ← contá-los

- 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...

```
/**
 * Algum aluno passa?
 *
 * @return true se algum aluno passa
 */
public boolean alguemPassa() {

    return lstAlunos.stream().anyMatch(a -> a.passa());
}
```

```
/**
 * Algum aluno passa?
 *
 * @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...

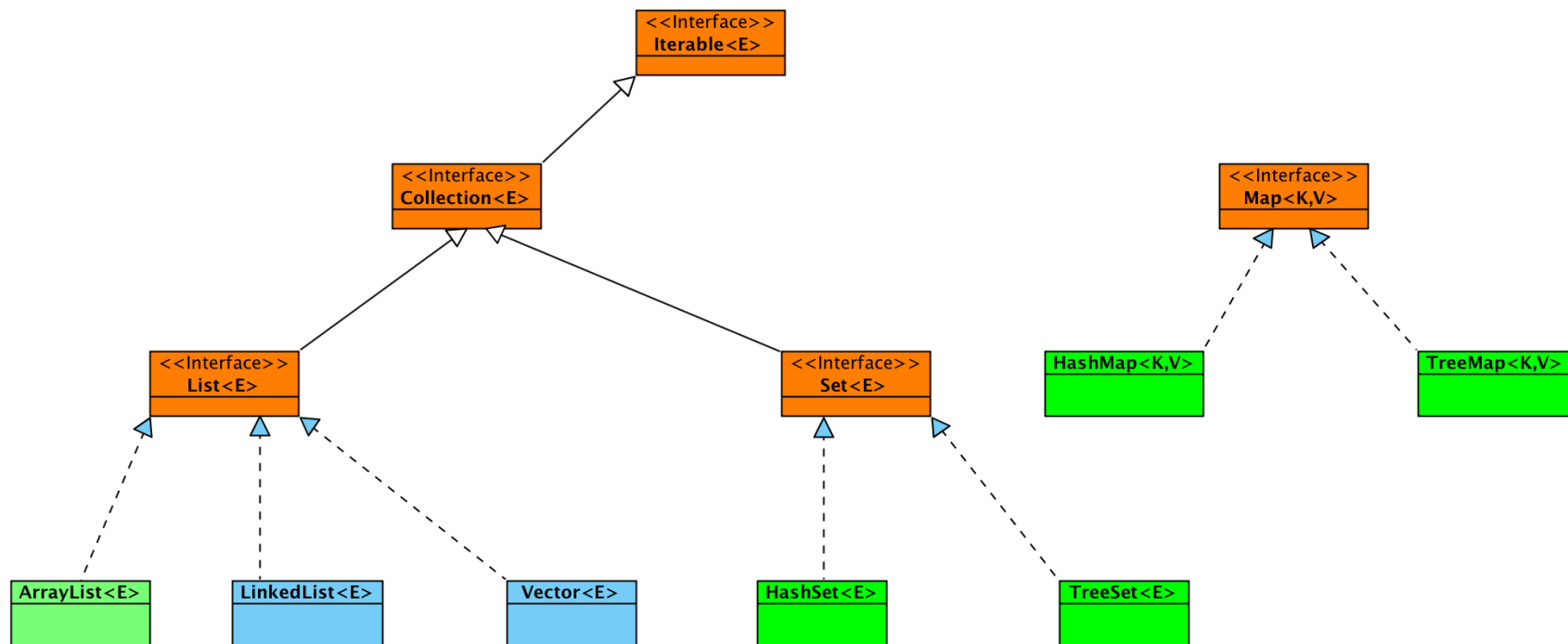
```
/**
 * Média da turma
 *
 * @return um double com a média da turma
 */
public double media() {
    double tot = lstAlunos.stream()
                          .mapToDouble(Aluno::getNota)
                          .sum();
    return tot/lstAlunos.size();
}
```

```
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>

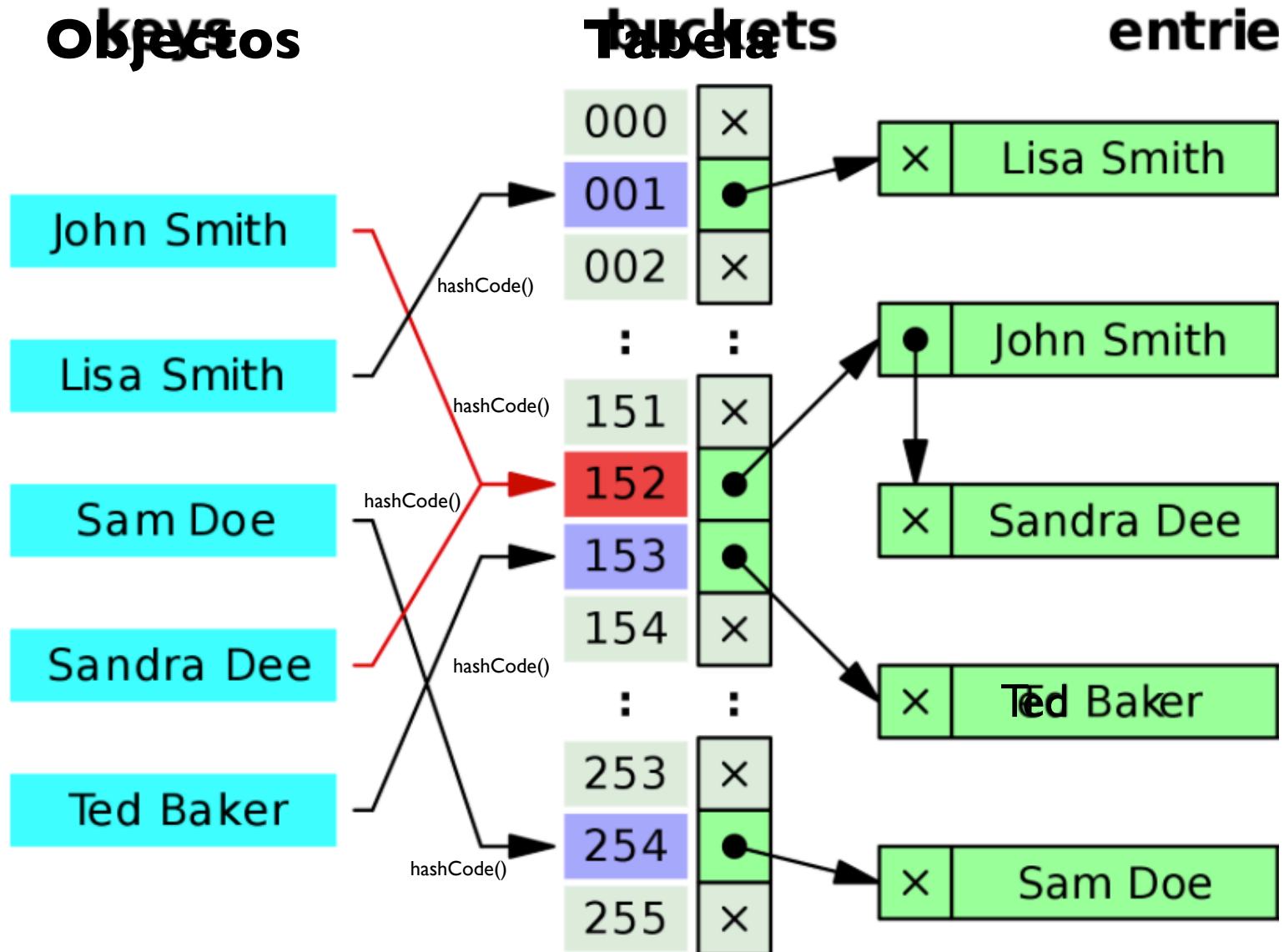
Set<E>

- 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()

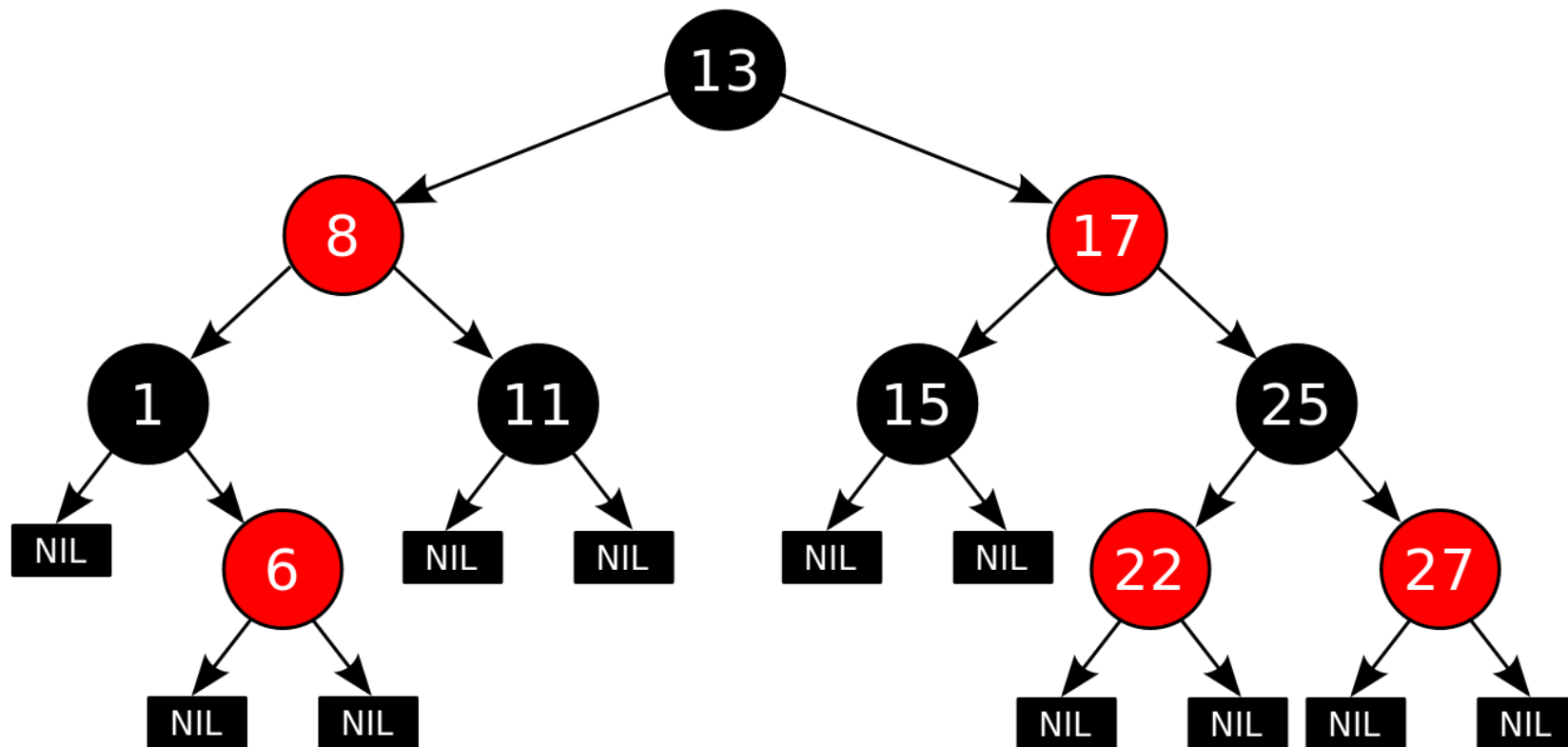
(exemplo!)

1. Definir **int hash = x ;** //(x diferente de 0)
2. 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.
3. Combinar cada um dos valores calculados acima no resultado do seguinte modo: **hash = 37 * hash + valor;**
4. **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

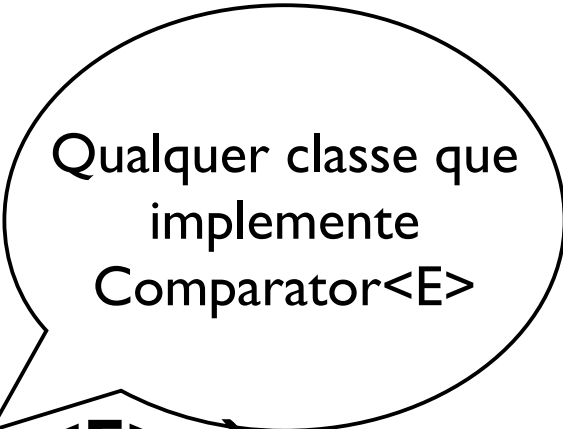
```
public int compareTo(Aluno a) {  
    return this.nome.compareTo(a.getNome());  
}
```

- 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);
```