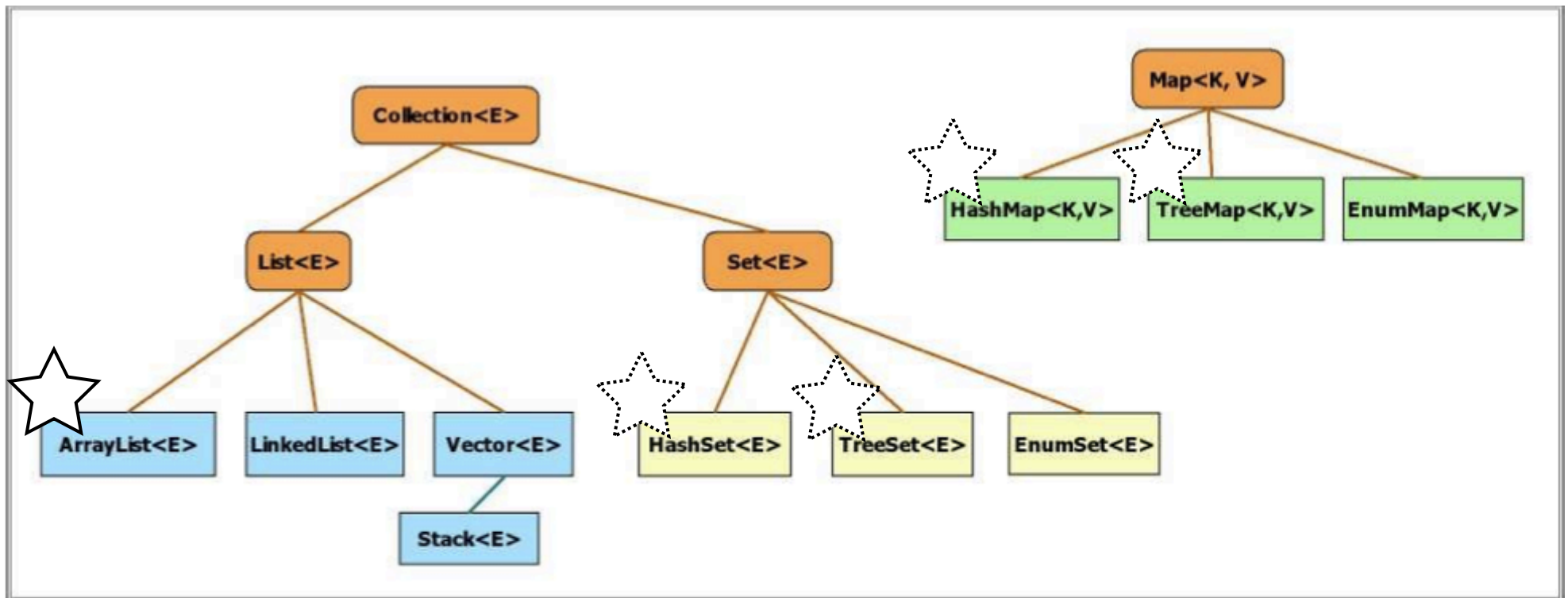


# Collecções e Maps



# 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&lt;E&gt; iterator()</code>
Iteradores internos	<code>Stream&lt;E&gt; stream()</code> <code>void forEach(Consumer c)</code>
Outros	<code>boolean equals(Object o)</code> <code>int hashCode()</code>

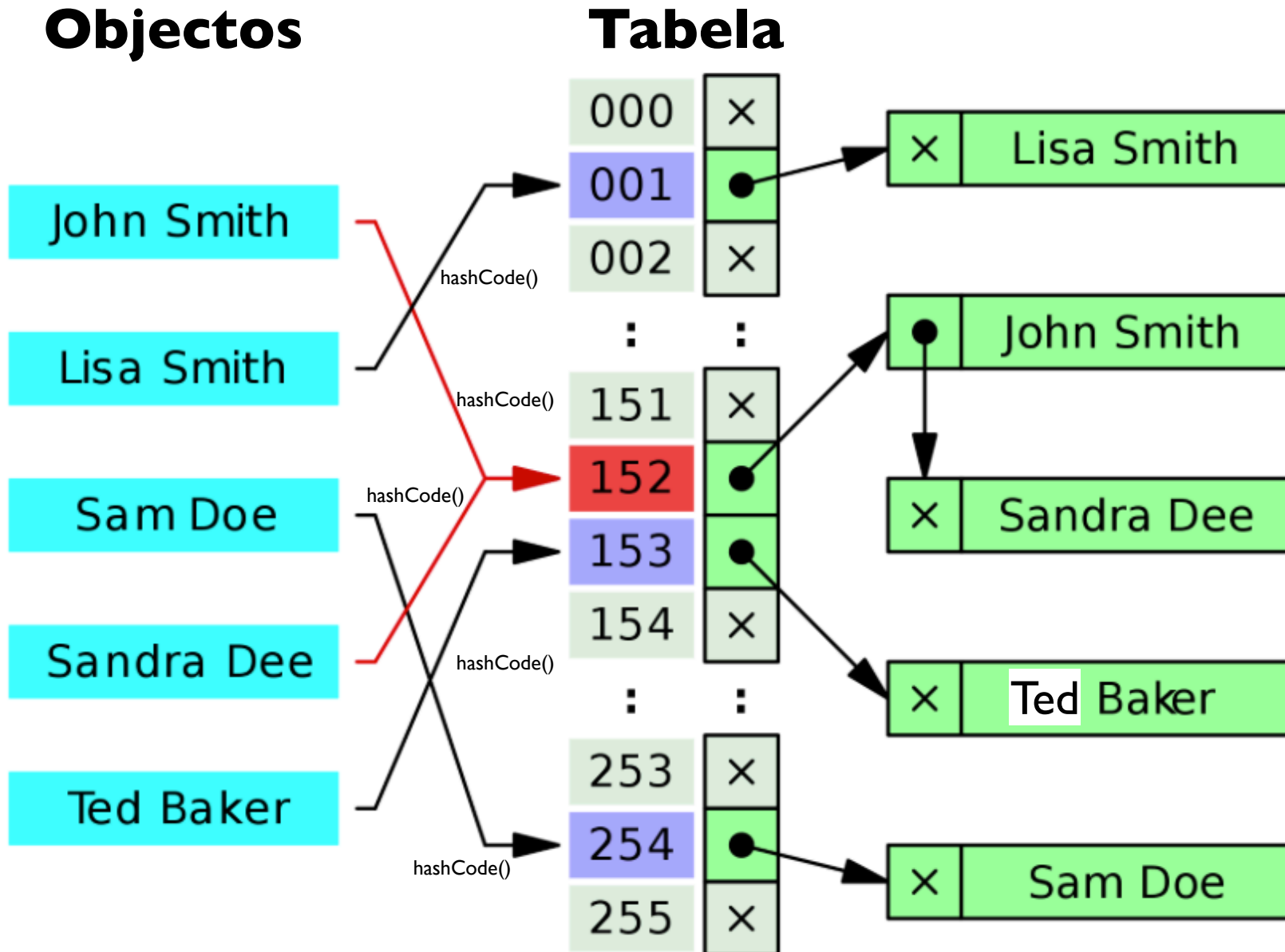
# Set<E>

- 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 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
  - recorre à referência do objecto
  - objectos iguais 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()

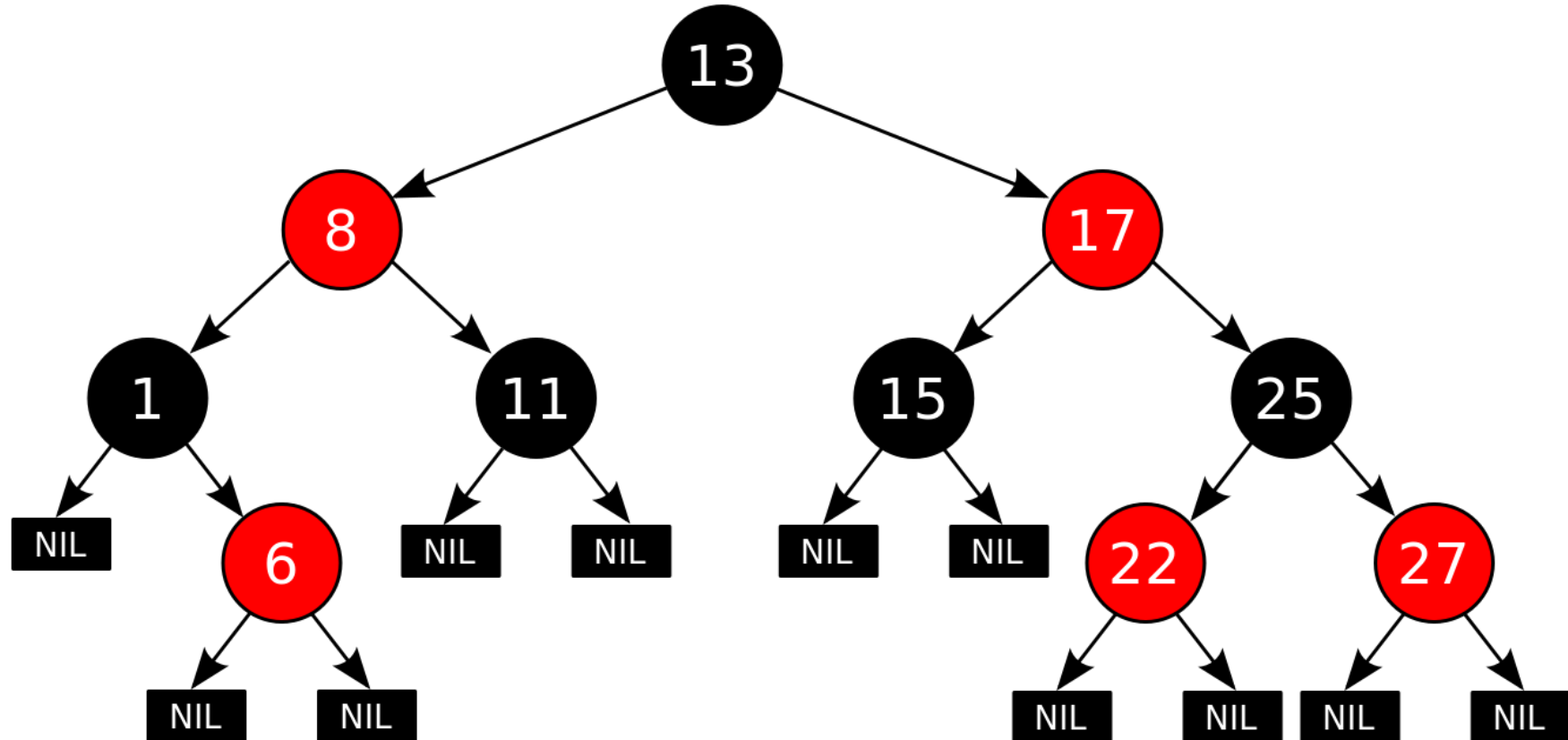
1. Definir **int hash = x ;** //(x diferente de 0)
2. Calcular o código de hash de cada var. instância **v** conforme o seu tipo:
  - boolean: **(v ? 0 : 1);**
  - byte, char, short ou int: **(int)v;**
  - long: **(int)(v ^ (v >>> 32));**
  - float: **Float.floatToIntBits(v);**
  - double: calcular **Double.doubleToLongBits(v)** 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
- Compara o objecto receptor com outro passado como parâmetro
- Se objectos são iguais
  - resultado: **0**
- Se objecto receptor é “maior”
  - resultado: **1**
- Se objecto receptor é “menor”
  - resultado: **-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

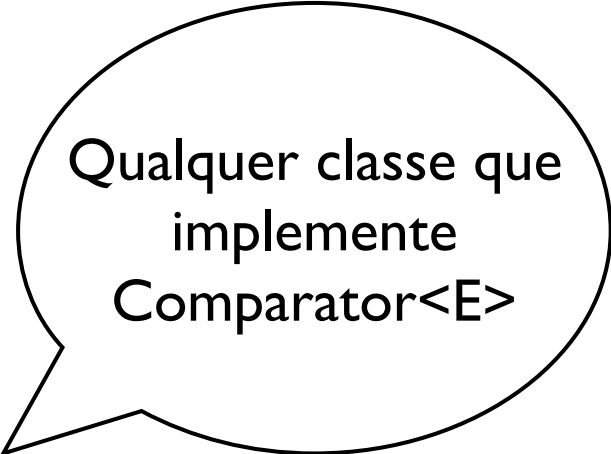
```
public int compareTo(Aluno a) {  
    return this.nome.compareTo(a.getNome());  
}
```

- 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**



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 por sua vez implementar
- 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.

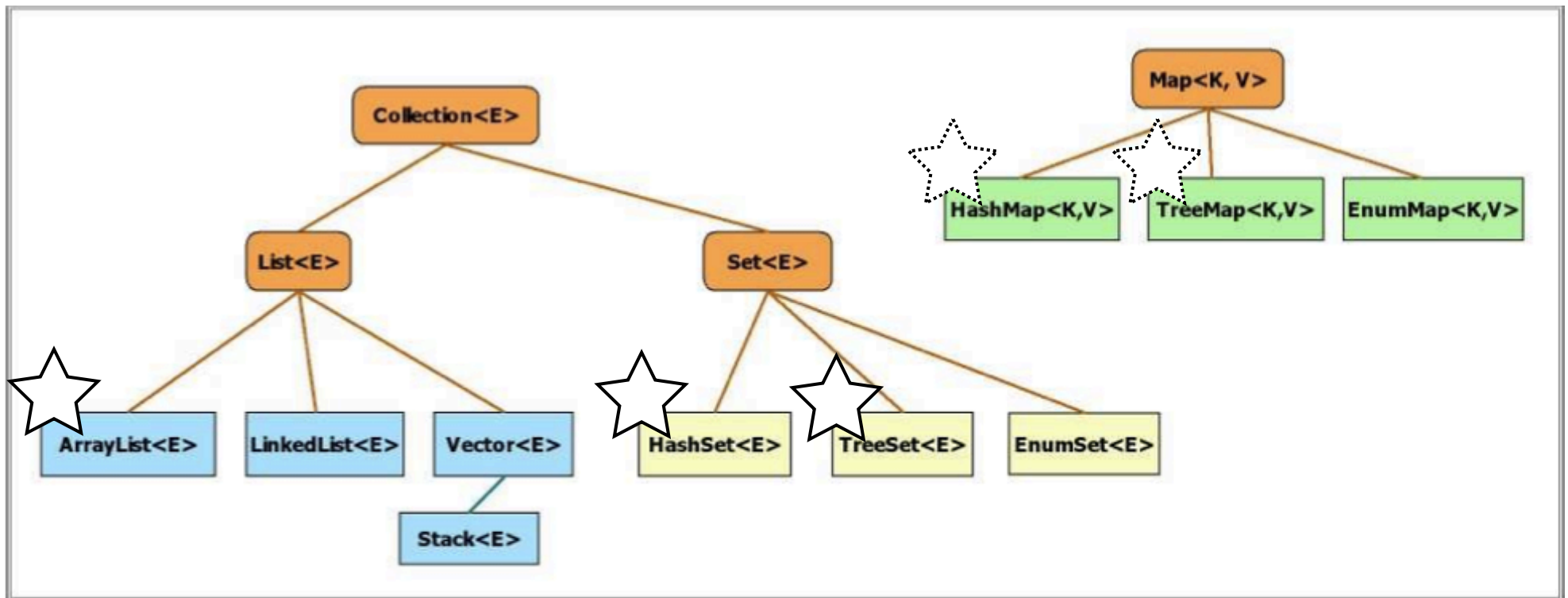
boolean

`equals(Object obj)`

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



# Collecções 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&lt;V&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<V> keySet()`**
  - Conjuntos das chaves
- **`Collection<V> values()`**
  - Coleção 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).

# TreeMap<K,V>

## TreeMap<K, V> métodos adicionais

`TreeMap<K, V>()`

`TreeMap<K, V>(Comparator<? super K> c)`

`TreeMap<K, V>(Map<? extends K, ? extends V> m)`

`K firstKey()`

`SortedMap<K, V> headMap(K toKey)`

`K lastKey()`

`SortedMap<K, V> subMap(K fromKey, K toKey)`

`SortedMap<K, V> tailMap(K fromKey)`

# Collecções e Maps

