Computação Orientada a Objetos

Coleções Java Parte V

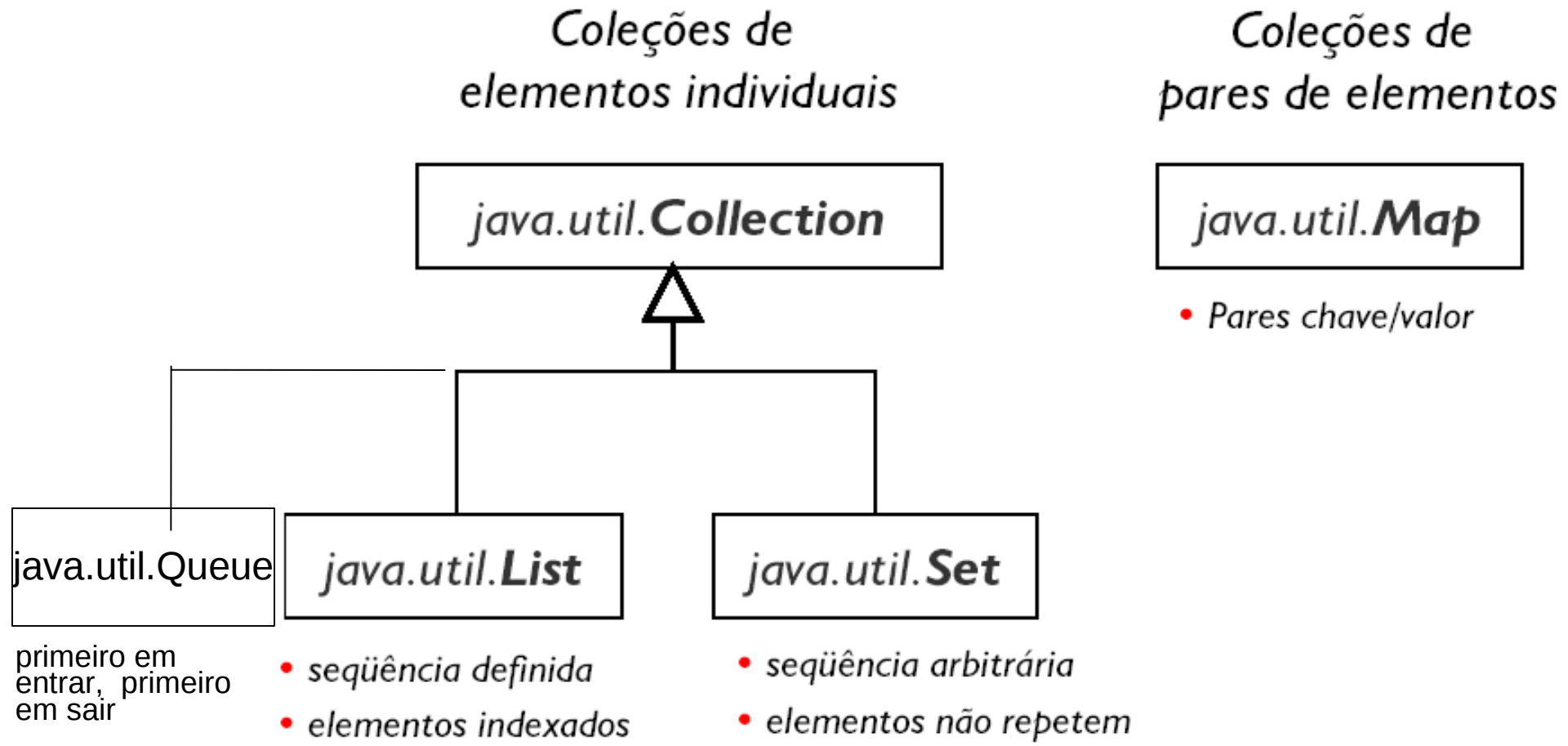
Slides baseados em:
Deitel, H.M.; Deitel P.J. Java: Como Programar, Pearson Prentice Hall, 6a Edição, 2005. Capítulo 19

Profa. Karina Valdivia Delgado
EACH-USP

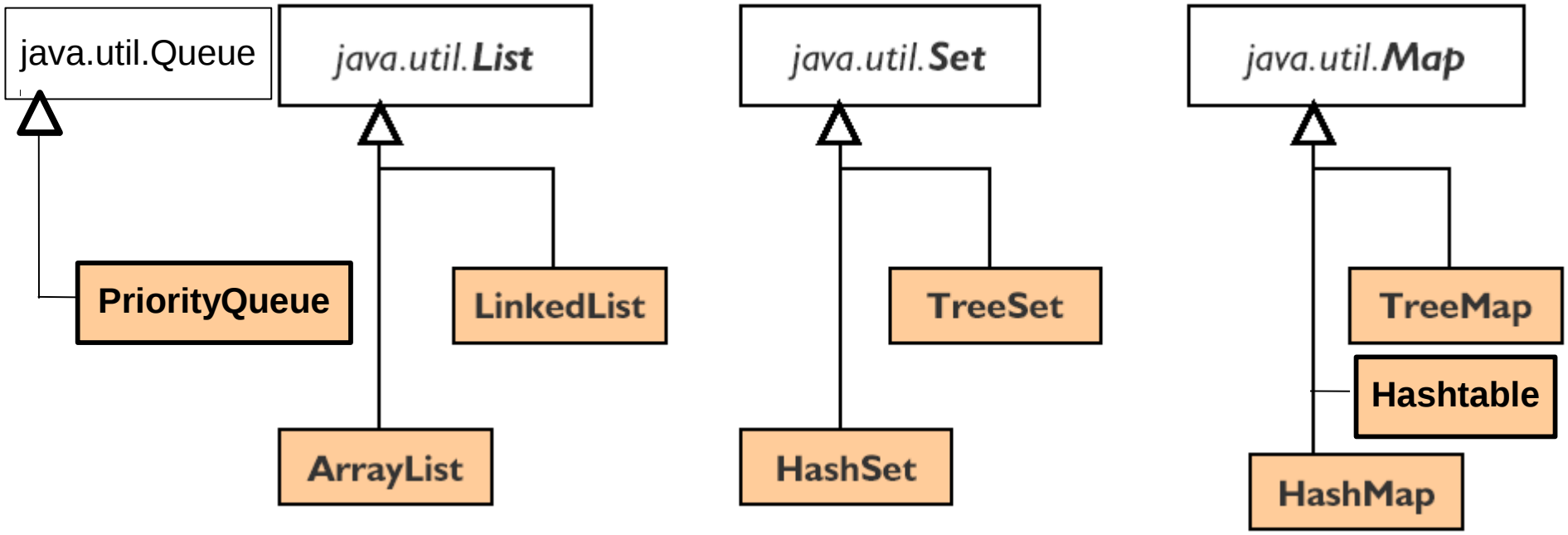
Revisando: O que é uma coleção?

- É uma estrutura de dados (um objeto) que agrupa referências a vários outros objetos.

Interfaces da estrutura de coleções



Implementações da estrutura de coleções



Interface **`Collection`**

- Operações básicas:
  - adiciona elemento: **`add (Object o)`**
  - remove elemento:**`remove (Object o)`**
- Operações de volume:
  - adiciona coleção: **`addAll (Collection c)`**
  - remove coleção: **`removeAll (Collection c)`**
  - mantém coleção: **`retainAll (Collection c)`**
  - `remove todos os elementos:` **`clear()`**
- retorna um objeto **`Iterator`** para percorrer a coleção: **`iterator()`**
- **`int`** **`size()`**
- **`boolean`** **`isEmpty()`**
- **`boolean`** **`contains (Object o)`**

Interface `List`

- `Fornece adicionalmente métodos para:`
  - manipular elementos via seus índices. Ex:
    - add(int index,Object o): Adiciona elemento. O tamanho da lista aumenta em 1.
    - remove(int index): Remove elemento da posição especificada e move todos os elementos após o elemento removido diminuindo o tamanho da lista em 1.
    - set(int index, Object o): Substitui elemento. O tamanho da lista permanece igual.
  - manipular um intervalo específico de elementos. Ex:
    - addAll(int index,Collection c) : Insere na posição especificada
    - subList(int fromIndex,int toIndex): obtem uma parte da lista, o indice final não faz parte do intervalo. Qualquer alteração na sublista também será feita na lista original (view)
  - recuperar elementos
    - get(int index)
  - obter um `ListIterator` para percorrer a lista.

Interface **Iterator**

Essa interface permite ver qualquer coleção como uma estrutura sequencial

- Determinar se a coleção tem mais elementos: hasNext()
- Obter uma referência ao próximo elemento da coleção: next()
- Apagar o último item retornado pelo método next(): remove()

Interface List**Iterator**

- Fornece adicionalmente os seguintes métodos:
  - determinar se há mais elementos ao percorrer a lista em ordem invertida: hasPrevious()
  - Obter uma referência ao elemento anterior da lista: previous()
  - para substituir o último item retornado pelo método next()ou previous(): set(Object o)
  - adiciona um objeto na posição atualmente apontada pelo iterador: add(Object o)

Classe **Collections**

- Exemplos de algoritmos que operam em objetos do tipo **`List`**:
  - **`sort:`** classifica os elementos da lista

```
Collections.sort( list);
```

  - **`binarySearch:`** localiza um elemento da lista

```
int result=Collections.binarySearch(list,key);
```

  - **`reverse:`** inverte os elementos da lista

```
Collections.reverse( list);
```

  - **`shuffle:`** ”embaralha” os elementos da lista

```
Collections.shuffle(list );
```

Classe **Collections**

- Exemplos de algoritmos que operam em objetos do tipo **`Collection`**:
  - **`min:`** retorna o menor elemento em uma coleção.
  - **`max:`** retorna o maior elemento em uma coleção.
  - **`frequency:`** calcula quantos elementos em uma coleção são iguais a um elemento especificado.

```
int result=Collections.frequency(list,key);
```

  - **`disjoint:`** determina se duas coleções não têm nenhum elemento em comum.

```
boolean result=Collections.disjoint(list1,list2);
```

List x Set

- A interface **List** permite elementos duplicados, enquanto **Set** define um conjunto de elementos únicos.
- **List** mantem a ordem em que os elementos foram adicionados.
- A busca em um **Set** pode ser mais rápida do que em um objeto do tipo **List**
- Um objeto do tipo **List** possui os elementos indexados

Classe TreeSet

- Alguns métodos:
  - headSet(Object e): obter um subconjunto (uma "view") em que cada elemento é menor do que o Elemento e
  - tailSet(Object e): obter um subconjunto (uma "view") em que cada elemento é maior ou igual do que o Elemento e
  - first(): obter o primeiro elemento do conjunto
  - last(): obter o último elemento do conjunto

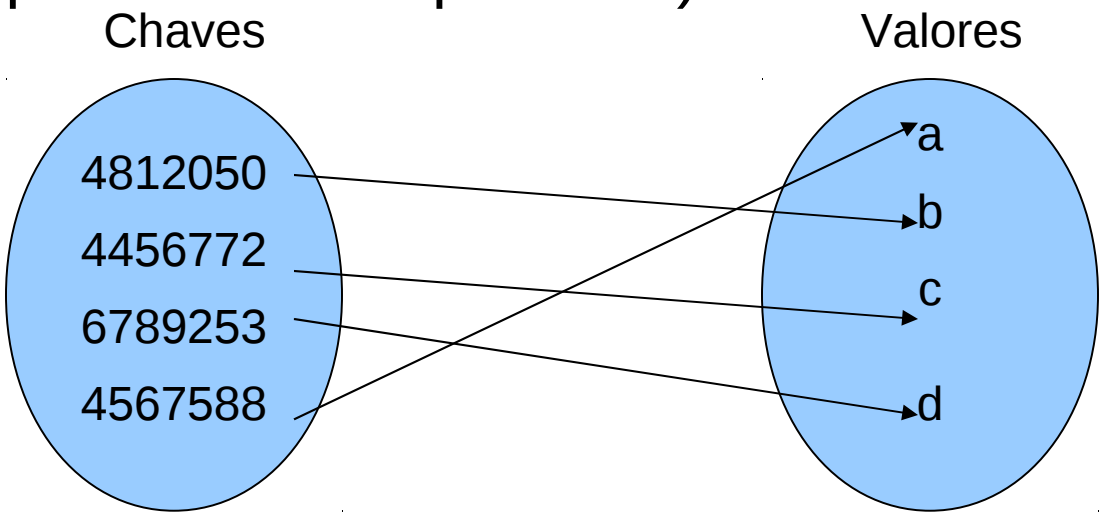
TreeSet x HashSet

- Um **TreeSet** utiliza uma árvore rubro-negra como implementação
- Um **HashSet** usa uma tabela de espalhamento como implementação
- Um **TreeSet** gasta computacionalmente O(log(n)) para inserir, enquanto o **HashSet** gasta apenas O(1).

Mapas

Mapas

- Um mapa (**`Map`**) associa chaves a valores e não pode conter chaves duplicatas
  - cada chave pode mapear somente um valor (mapeamento um para um)



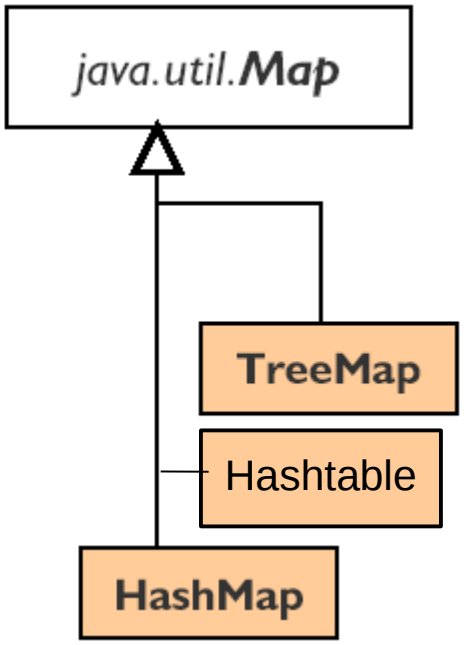
Interface **Map**

- Operações básicas:
  - adiciona par: **`put (Object key,Object value)`**
  - `devolve valor associado à chave`**`:get(Object key)`**
  - remove par: **`remove (Object key)`**
- Operações de volume:
  - adiciona mapa: **`putAll (Map m)`**
  - obtem o conjunto de chaves: **`keySet()`**
  - obtem a coleção de valores: **`values()`**
  - `remove todos os pares:` **`clear()`**
- **`int size()`**
- **`boolean isEmpty()`**
- **`boolean containsKey (Object key)`**
- **`boolean containsValue (Object value)`**

Mapas

Classes que implementam a interface **`Map`**

- **`HashMap`**
- **`TreeMap`**
- **`Hashtable`**



HashMap – Exemplo 1

```
Map<Integer,String> mapa=new Hashmap<Integer, String>();
mapa.put(455,"vermelho");
mapa.put(333,"branco");
mapa.put(678,"amarelo");
mapa.put(455,"azul");
System.out.println(mapa);
```

– Saída do programa:

```
{455=azul, 678=amarelo, 333=branco}
```

HashMap – Exemplo 1

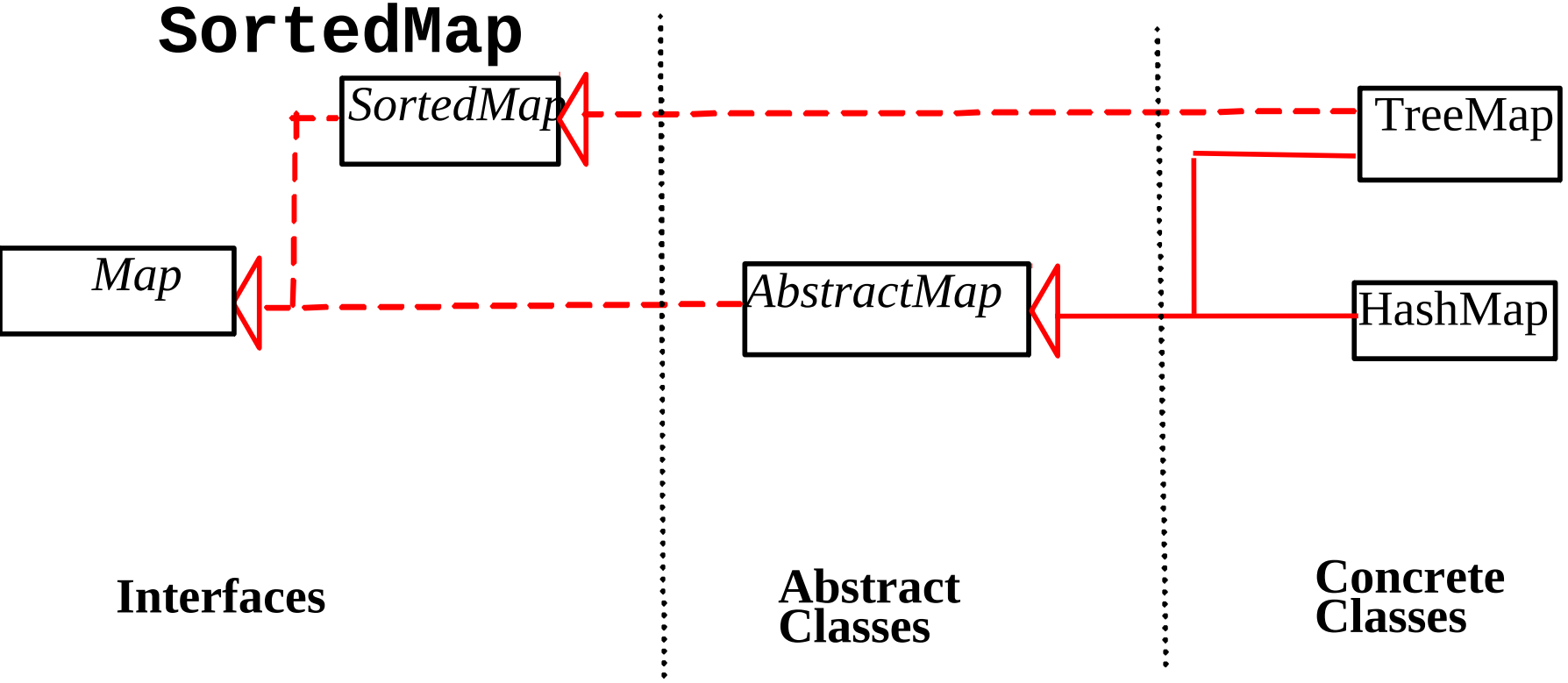
```
Map<Integer, String > mapa = new Hashmap<Integer, String>();
mapa.put(455,"vermelho");
mapa.put(333,"branco");
mapa.put(678,"amarelo");
mapa.put(455,"azul");
System.out.println(mapa.keySet());
System.out.println(mapa.values());
```

– Saída do programa:

```
[455, 678, 333]
[azul, amarelo, branco]
```

Mapas ordenados

- A interface **`SortedMap`** estende a interface **`Map`** e mantém as suas chaves ordenadas
- A classe **`TreeMap`** implementa a interface **`SortedMap`**



Classe TreeMap

- Alguns métodos:
  - headMap(Object key): obter um subconjunto (uma "view") do mapa em que cada par tem chave menor do que key
  - tailMap(Object key): obter um subconjunto (uma "view") do mapa em que cada par tem chave maior ou igual do que key
  - firstKey(): obter a primeira chave do mapa
  - lastKey(): obter a última chave do mapa

TreeMap – Exemplo 1

```
Map<Integer, String > mapa = new TreeMap<Integer, String>();
mapa.put(455,"vermelho");
mapa.put(333,"branco");
mapa.put(678,"amarelo");
mapa.put(455,"azul");
System.out.println(mapa);
```

- Saída do programa:

```
{333=branco, 455=azul, 678=amarelo}
```

TreeMap – Exemplo 1

```
Map<Integer, String > mapa = new TreeMap<Integer, String>();
mapa.put(455,"vermelho");
mapa.put(333,"branco");
mapa.put(678,"amarelo");
mapa.put(455,"azul");
System.out.println(mapa);
System.out.println(mapa.firstKey());
System.out.println(mapa.lastKey());
System.out.println(mapa.headMap(455));
System.out.println(mapa.tailMap(455));
```

```
{333=branco, 455=azul, 678=amarelo}
333
678
{333=branco}
{455=azul, 678=amarelo}
```

TreeSet e TreeMap

- A ordem nessas estruturas de dados é definida pelo método de comparação entre seus elementos.
- Opção 1:
  - Fazer com que a classe dos elementos se torne "comparável" implementando a interface **Comparable** e incluindo o método **compareTo**

TreeSet e TreeMap

- A ordem nessas estruturas de dados é definida pelo método
elementos.
- Opção 1:
  - Fazer com que a classe dos elem orne "comparável" implementando a interface **omparable** e incluindo o método **compareTo**

Este método deve retornar **zero**, se o objeto comparado for igual a este objeto, um número **negativo**, se este objeto for menor que o objeto dado, e um número **positivo**, se este objeto for maior que o objeto dado.

TreeSet e TreeMap

- A ordem nessas estruturas de dados é definida pelo método de comparação entre seus elementos.
- Opção 1:
  - Fazer com que a classe dos elementos se torne "comparável" implementando a interface **Comparable** e incluindo o método **compareTo**
- Opção 2:
  - Criar uma classe que implementa a interface **Comparator**, e incluir o método **compare**.

TreeSet e TreeMap

- A ordem nessas estruturas de dados é definida pelo método de comparação entre seus elementos.
- Opção 1:
  - Fazer com que
  “comparável” imp
  e incluindo o mé

Esse método compara dois objetos e retorna um inteiro **negativo** se o primeiro for menor do que o segundo; **zero**, se forem idênticos; e um valor **positivo**, caso contrário.

- Opção 2:
  - Criar uma classe que implementa a terface **Comparator**, e incluir o método **compare**.

Comparable

```
public class Employee implements Comparable{
    private int id;
    private String name;
    public Employee(int i, String n){
          this.id=i;
          this.name=n;
    }
    public int getId(){
          return id;
    }
    public String getName(){
          return name;
    }
     public int compareTo(Object obj){
          int returnValue;
          if(this.id==((Employee) obj).getId())
              returnValue=0;
          else
            if(this.id>((Employee) obj).getId())
                returnValue=1;
            else
                returnValue=-1;
          return returnValue;
     }
}
```

Comparator

```
public class Employee1{
    private int id;
    private String name;
    public Employee1(int i, String n){
        this.id=i;
        this.name=n;
    }
    public int getId(){
        return id;
    }
    public String getName(){
        return name;
    }
}
```

Comparator

```
public class EmployeeComparator implements Comparator
<Employee1>{

     public int compare(Employee1 o1, Employee1 o2) {
          int returnValue;
          if(o1.getId()==o2.getId())
                returnValue=0;
          else
              if(o1.getId()>o2.getId())
                  returnValue=1;
              else
                  returnValue=-1;
          return returnValue;
      }
}
```

HashSet e HashMap

- Usam tabela hash
- Para adicionar um objeto a uma tabela hash é calculado o **hashCode** do objeto.
- Para que isso funcione corretamente é necessário verificar que o método **hashCode** de cada objeto retorne o mesmo valor para dois objetos, se eles são considerados iguais:

Se **a.equals(b)** implica **a.hashCode() = b.hashCode()**

HashSet e HashMap

Para a classe Employee com:

```
private int id;
private String name;
```

Incluimos os métodos hashCode e equals

```
public int hashCode() {
  final int PRIME = 31;
  int result = 1;
  result = PRIME * result + id;
  result = PRIME * result + ((name == null) ? 0 : name.hashCode());
  return result;
  }
```

HashSet e HashMap

Para a classe Employee com:

```
private int id;
private String name;
```

Podemos usar o Eclipse para isso: Ir no menu **Source** opção **Generate hashcode() and equals()**.

Incluimos os métodos hashCode e equals

```
public int hashCode() {
  final int PRIME = 31;
  int result = 1;
  result = PRIME * result + id;
  result = PRIME * result + ((name == null) ? 0 : name.hashCode());
  return result;
  }
```

HashSet e HashMap

```
public boolean equals(Object obj) {
        if (this == obj)
                return true;
        if (obj == null)
                return false;
        if (getClass() != obj.getClass())
                return false;
        final Employee other = (Employee) obj;
        if (id != other.id)
                return false;
        if (name == null) {
                if (other.name != null)
                        return false;
        } else if (!name.equals(other.name))
                return false;
        return true;    }
```

Exercício

```
Entre com uma frase:
To be or not to be: that is the question
Saída:
Key             Value
be              1
be:             1
is              1
not             1
or              1
question        1
that            1
the             1
to              2
```

Classe `StringTokenizer`

Essa classe permite dividir a string de entrada em tokens. Um token é uma parte de um string.

Ex:

```
String s = "Five Three Nine One"
StringTokenizer st = new StringTokenizer(s);
```

Métodos:

- Determinar se a string tem mais tokens: hasMoreTokens()
- Obter uma referência ao próximo token da string: nextToken()
- Determinar o número de tokens na string: countTokens()

Coleção HashMap - Exemplo

Utiliza uma coleção **HashMap** para contar o número de ocorrências de cada palavra em uma string

```
import java.util.StringTokenizer;
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.TreeSet;
import java.util.Scanner;

public class WordTypeCount{
       private Map< String, Integer > map;
       private Scanner scanner;
       public WordTypeCount(){
            map = new HashMap< String, Integer >(); // cria HashMap
            scanner = new Scanner( System.in ); // cria scanner
            createMap(); // cria mapa baseado na entrada do usuário
            displayMap(); // apresenta conteúdo do mapa
        }//fim do construtor de WordTypeCount
```

Coleção HashMap - Exemplo

```
import java.util.StringTokenizer;
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.TreeSet;
import java.util.Scanner;

pub
        public WordTypeCount(){
             map = new HashMap< String, Integer >(); // cria HashMap
             scanner = new Scanner( System.in ); // cria scanner
             createMap(); // cria mapa baseado na entrada do usuário
             displayMap(); // apresenta conteúdo do mapa
}//fim do construtor de WordTypeCount
```

Cria uma coleção **HashMap** vazia com chaves do tipo **String** e valores do tipo **Integer**

Coleção HashMap - Exemplo

```
import java.util.StringTokenizer;
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.TreeSet;
import java.util.Scanner;

public
                map = new HashMap< String, Integer >(); // cria HashMap
                scanner = new Scanner( System.in ); // cria scanner
                createMap(); // cria mapa baseado na entrada do usuário
                displayMap(); // apresenta conteúdo do mapa
}//fim do construtor de WordTypeCount
```

Cria uma objeto **Scanner** que lê a entrada do usuário a partir do fluxo de entrada padrão

Coleção HashMap - Exemplo

```
import java.util.StringTokenizer;
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.TreeSet;
import java.util.Scanner;

public class WordTypeCount{
```

Chama o método **createMap** para armazenar no mapa o número de ocorrências de cada palavra na frase

```
                HashMap< String, Integer >(); // cria HashMap
        scanner = new Scanner( System.in ); // cria scanner
        createMap(); // cria mapa baseado na entrada do usuário
        displayMap(); // apresenta conteúdo do mapa
}//fim do construtor de WordTypeCount
```

Coleção HashMap - Exemplo

```
import java.util.StringTokenizer;
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.TreeSet;
import java.util.Scanner;

public class WordTypeCount{
        private Map< String, Integer > map;
```

Chama o método **displayMap** para exibir todas as entradas do mapa

```
                                                // cria HashMap
                                                // cria scanner
            createMap(); // cria mapa baseado na entrada do usuário
            displayMap(); // apresenta conteúdo do mapa
}//fim do construtor de WordTypeCount
```