

# Estruturas de Dados

## Java Collections

UA.DETI.POO

# JAVA Collections Framework (JCF)

---

- ❖ Conjunto de classes, interfaces e algoritmos que representam vários tipos de estruturas de armazenamento de dados
  - Listas, Vectors, Pilhas, Árvores, Mapas,...
  - Permitem agregar objetos de um tipo paramétrico - os tipos de dados também são um Parâmetro
  - Exemplo:

```
ArrayList<String> cidades = new ArrayList<>();  
cidades.add("Aveiro");  
cidades.add("Paris");
```
  - Não suportam tipos primitivos (int, float, double,...). Neste caso, precisamos de usar classes adaptadoras (Integer, Float, Double, ...)

# Exemplo

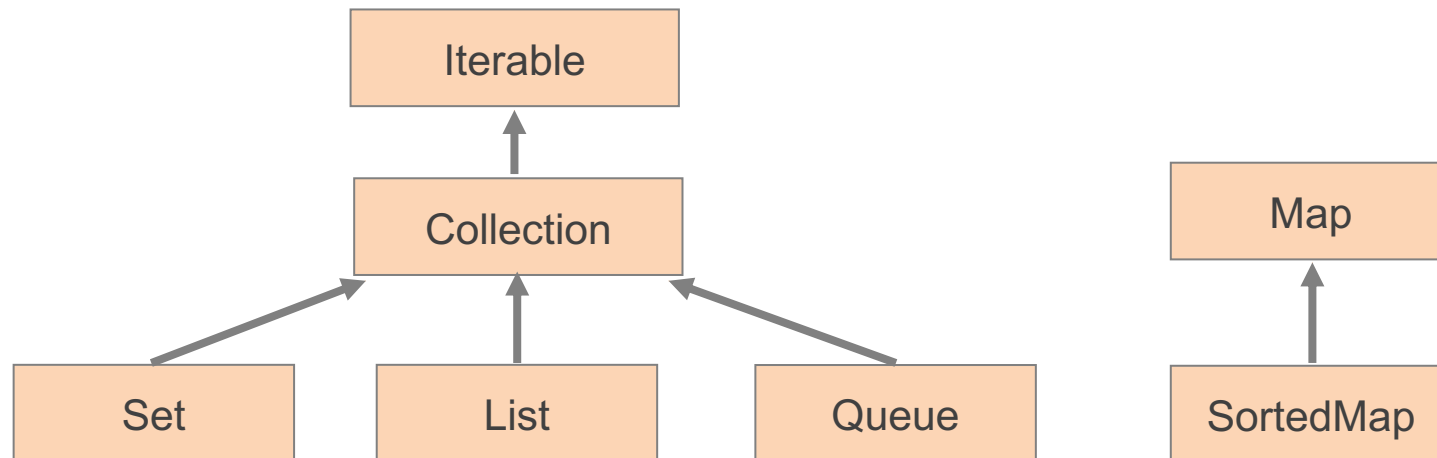
```
public class CreateArrayListExample {  
    public static void main(String[] args) {  
        // Creating an ArrayList of String  
        List<String> animals = new ArrayList<>();  
        // Adding new elements to the ArrayList  
        animals.add("Lion");  
        animals.add("Tiger");  
        animals.add("Cat");  
        animals.add("Dog");  
        System.out.println(animals);  
        // Adding an element at a particular index in an ArrayList  
        animals.add(2, "Elephant");  
        System.out.println(animals);  
        // Find the index of the first occurrence of an element  
        System.out.println(animals.indexOf("Cat"));  
    }  
}
```

```
[Lion, Tiger, Cat, Dog]  
[Lion, Tiger, Elephant, Cat, Dog]  
3
```

# Principais Interfaces

## ❖ Conjunto de 4 Interfaces Principais:

- Conjuntos (**Set**): sem noção de posição (sem ordem), sem repetição
- Listas (**List**): sequências com noção de ordem, com repetição
- Filas (**Queue**): são as filas do tipo *First in First Out*
- Mapas (**Map**): estruturas associativas onde os objectos são representados por um par chave-valor.



# Vantagens das Collections

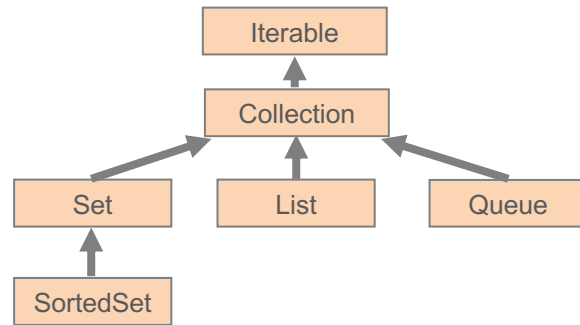
---

- ❖ Vantagem de criar interfaces:
  - Separa-se a especificação da implementação
  - Pode-se substituir uma implementação por outra mais eficiente sem grandes impactos na estrutura existente.

- ❖ Exemplo:

```
Collection<String> c = new LinkedList<>();  
c.add("Aveiro");  
c.add("Paris");  
Iterator<String> i = c.iterator();  
while (i.hasNext()) {  
    System.out.println(i.next());  
}
```

# Expansão de contratos



```
<<interface>>
Collection<E>

+add(E):boolean
+remove(Object):boolean
+contains(Object):boolean
+size():int
+iterator():Iterator<E> etc...
```

```
<<interface>>
List<E>

+add(E):boolean
+remove(Object):boolean
+get(int):E
+indexOf(Object):int
+contains(Object):boolean
+size():int
+iterator():Iterator<E>
etc...
```

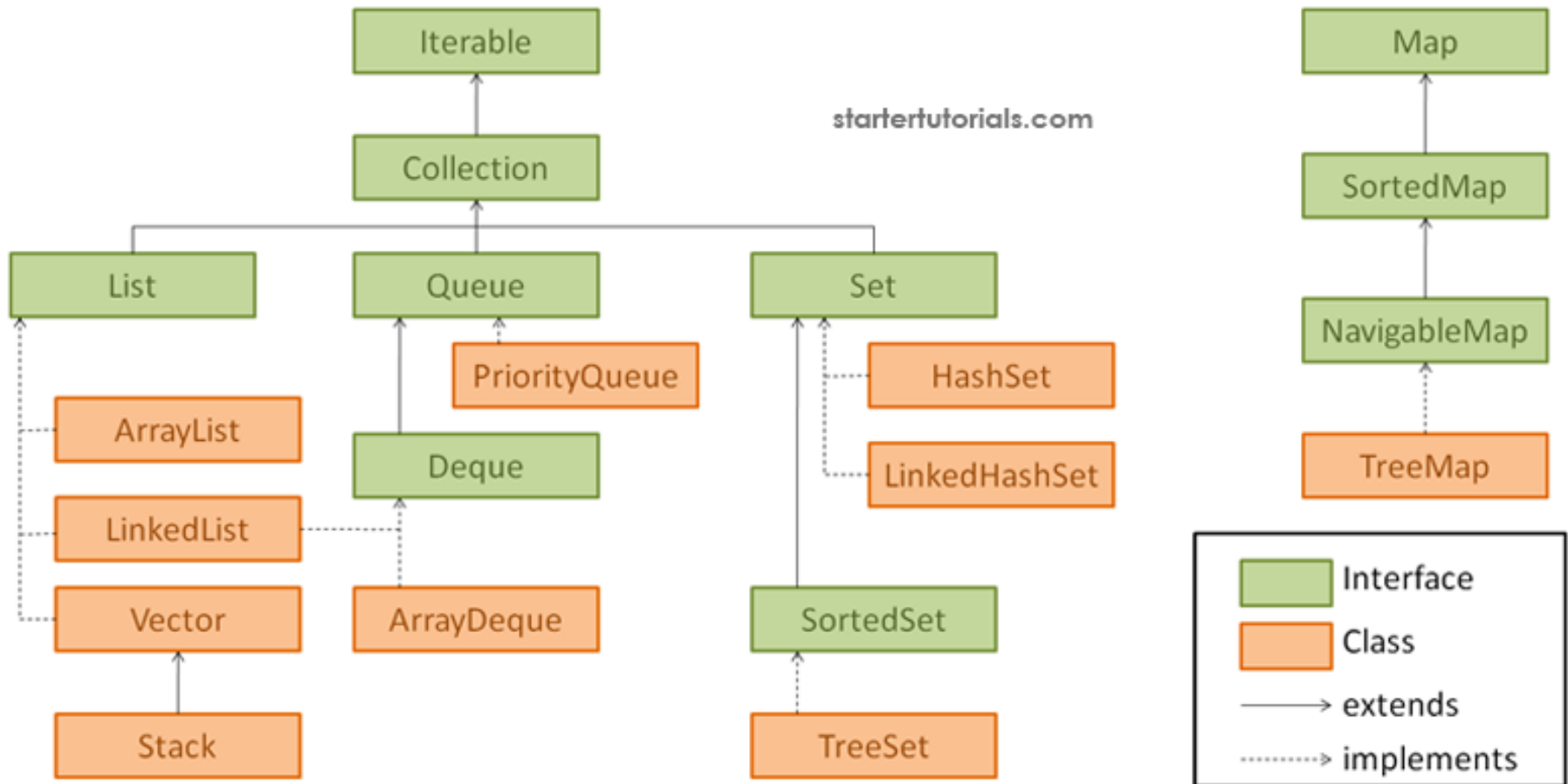
```
<<interface>>
Set<E>

+add(E):boolean
+remove(Object):boolean
+contains(Object):boolean
+size():int
+iterator():Iterator<E> etc...
```

```
<<interface>>
SortedSet<E>

+add(E):boolean
+remove(Object):boolean
+contains(Object):boolean
+size():int
+iterator():Iterator<E>
+first():E
+last():E
etc...
```

# Hierarquia de Classes



# Interfaces e Implementações

Collections					
	Implementações				
Interfaces	Resizable array	Linked list	Hash table	Hash table + Linked list	Balanced Tree (sorted)
List	ArrayList	LinkedList			
Queue	ArrayDeque	LinkedList			
Set			HashSet	LinkedHashSet	TreeSet
Map			HashMap	LinkedHashMap	TreeMap



# Interface Iterable

```
public interface Iterable<T> {
```

```
    default void forEach(Consumer<? super T> action)
```

```
    // Performs the given action for each element of the Iterable
```

```
    // until all elements have been processed or the action
```

```
    // throws an exception.
```

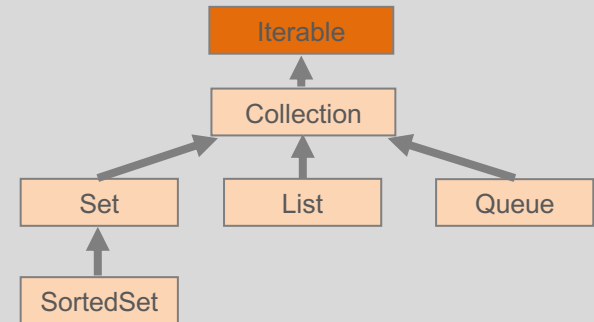
```
    Iterator<T> iterator()
```

```
    // Returns an iterator over elements of type T.
```

```
    default Spliterator<T> spliterator()
```

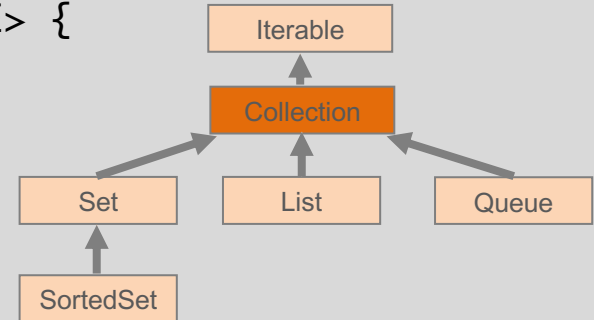
```
    // Creates a Spliterator over the elements described by this Iterable.
```

```
}
```



# Interface Collection

```
public interface Collection<E> extends Iterable<E> {  
    // Basic operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element); //optional  
    boolean remove(Object element); //optional  
    Iterator<E> iterator();  
    // Bulk operations  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c); //optional  
    boolean removeAll(Collection<?> c); //optional  
    boolean retainAll(Collection<?> c); //optional  
    void clear(); //optional  
    // Array operations  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
}
```

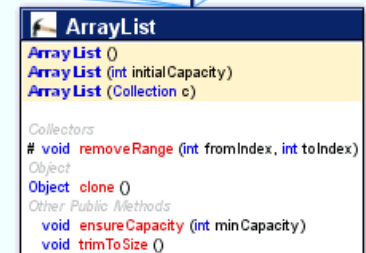
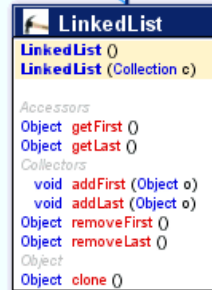
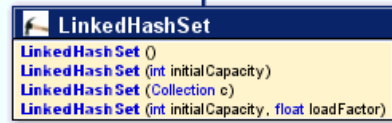
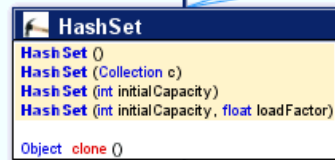
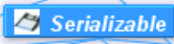
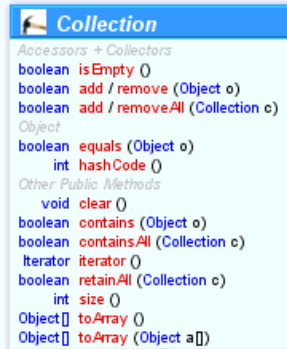


# java.util.\*

## Collection

Methods declared in Interfaces are hidden in subtypes

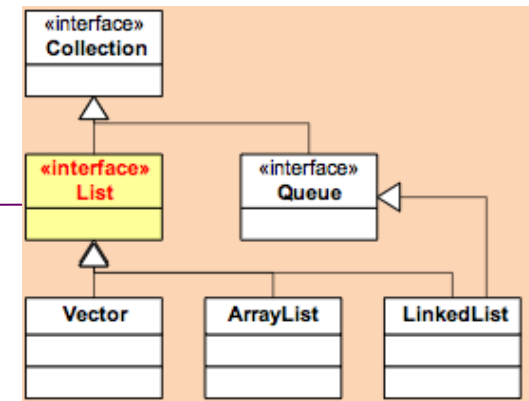
See also: [Legacy Collection Diagram](#)



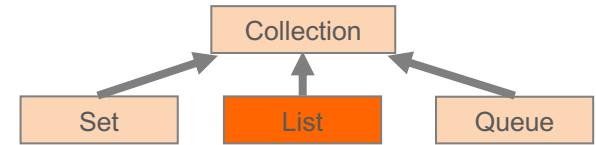
# Listas

- ❖ Implementam *List*
- ❖ Podem conter duplicados.
- ❖ Para além das operações herdadas de *Collection*, a interface *List* inclui ainda:
  - **Acesso Posicional** — manipulação de elementos baseada na sua posição (índice) na lista
  - **Pesquisa** — de determinado elemento na lista. Retorna a sua posição.
  - **ListIterator** — estende a semântica do *Iterator* tirando partido da natureza sequencial da lista.
  - **Range-View** — execução de operações sobre uma gama de elementos da lista.

`list.subList(fromIndex, toIndex).clear();`



# List



```
public interface List<E> extends Collection<E> {
    // Positional Access
    boolean add(E e)
    void add(int index, E element);           // Optional
    E get(int index);
    E set(int index, E element);             // Optional
    E remove(int index);                    // Optional
    boolean addAll(Collection<? extends E> c); // Optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(int index);

    // Range-view
    List<E> subList(int from, int to);
}
```



```
public interface ListIterator<E>
    extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove(); //optional
    void set(E e); //optional
    void add(E e); //optional
}
```

# Listas – Classes

---

Mais comuns:

- ❖ ArrayList – Array dinâmico
- ❖ LinkedList – Lista ligadas

Outras:

- ❖ Vector – Array dinâmico
  - (!) *Vector is synchronized. If a thread-safe implementation is not needed, it is recommended to use ArrayList in place of Vector.*
- ❖ Stack
  - extends Vector

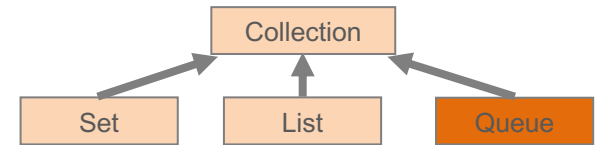
Diferenças?

# Listas – Exemplo

```
public static void main(String args[]) {  
    String[] str1 = {"Rui", "Manuel", "Jose", "Pires", "Eduardo", "Santos"};  
    String[] str2 = {"Rosa", "Pereira", "Rui", "Vidal", "Hugo", "Maria"};  
    List<String> larray = new ArrayList<>();  
    List<String> llist = new LinkedList<>();  
  
    for (String i: str1 ) larray.add(i);  
    for (String i: str2 ) llist.add(i);  
  
    llist.addAll(llist.size()/2, larray);  
    for (String ele: llist)  
        System.out.println( ele );  
  
    System.out.println("Rui está na posição " +  
        llist.indexOf("Rui") + " e " + llist.lastIndexOf("Rui"));  
  
    llist.set(llist.lastIndexOf("Rui"), "Rui2");  
    System.out.println(llist.lastIndexOf("Rui"));  
}
```

Rosa  
Pereira  
Rui  
Rui  
Manuel  
Jose  
Pires  
Eduardo  
Santos  
Vidal  
Hugo  
Maria  
Rui está na posição 2 e 3  
2

# Queue – Filas



```
public interface Queue<E> extends Collection<E> {  
  
    // Inserts the specified element in the queue  
    boolean offer(E e);  
  
    // Retrieves and removes the head of this queue  
    // throws an exception if empty  
    E remove();  
    // Retrieves and removes the head of this queue  
    E poll();  
  
    // Retrieves, but does not remove, the head of this queue  
    // throws an exception if empty  
    E element();  
    // Retrieves, but does not remove, the head of this queue  
    E peek();  
}
```



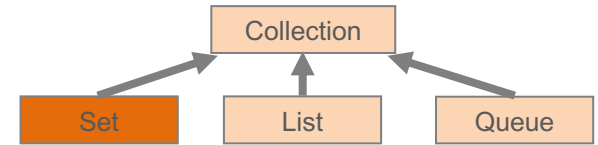


# Filas - Implementações

---

- ❖ `ArrayBlockingQueue`
- ❖ `ArrayDeque`
- ❖ `ConcurrentLinkedDeque`
- ❖ `ConcurrentLinkedQueue`
- ❖ `DelayQueue`
- ❖ `LinkedBlockingDeque`
- ❖ `LinkedBlockingQueue`
- ❖ `LinkedList`
- ❖ `LinkedTransferQueue`
- ❖ `PriorityBlockingQueue`
- ❖ `PriorityQueue`
- ❖ `SynchronousQueue`

# Set - Conjuntos



- ❖ Uma coleção que não pode conter elementos duplicados.
- ❖ Contém apenas os métodos definidos na interface *Collection*
  - Novos contratos nos métodos *add*, *equals* e *hashCode*
- ❖ Implementações:
  - HashSet
  - TreeSet organiza por ordem alfabética
  - ..

# AbstractSet

```
public abstract class AbstractSet<E> extends AbstractCollection<E>
    implements Set<E> {

    protected AbstractSet();

    public boolean equals(Object o) {
        if (!(o instanceof Set)) return false;
        return ((Set)o).size()==size() && containsAll((Set)o);
    }

    public int hashCode() {
        int h = 0;
        for( E el : this )
            if ( el != null ) h += el.hashCode();
        return h;
    }
}
```

# HashSet

---

- ❖ Usa uma tabela de dispersão (Hash Map) para armazenar os elementos.
- ❖ A inserção de um novo elemento não será efectuada se a função *equals* do elemento a ser inserido com algum elemento do Set retornar true.
  - É fundamental implementar a função *equals* em todas as classes que possam ser usadas como elementos de tabelas de dispersão (HashSet, HashMap,...)
- ❖ Desempenho constante,
  - $O(1)$  para add, remove, contains e size

```
java.lang.Object  
└ java.util.AbstractCollection<E>  
  └ java.util.AbstractSet<E>  
    └ java.util.HashSet<E>
```

# HashSet

```
public static void main(String args[]) {  
  
    // vector para simular a entrada de dados no Set  
    String[] str = {"Rui", "Manuel", "Rui", "Jose",  
                    "Pires", "Eduardo", "Santos"};  
  
    Set<String> group = new HashSet<>();  
    for (String i: str ) {  
        if (!group.add(i))  
            System.out.println("Nome duplicado: " + i);  
    }  
    System.out.println(group.size() + " nomes distintos");  
  
    for (String s: group)  
        System.out.println( s );  
}
```

**Nome duplicado: Rui**  
**6 nomes distintos**

Manuel  
Rui  
Jose  
Eduardo  
Santos  
Pires

**Ordem?**

**Conclusão:** sem noção de posição (sem ordem)

# TreeSet

---

- ❖ Permite a ordenação dos elementos pela sua “ordem natural”.
  - Os objetos inseridos em TreeSet devem implementar a interface Comparable.
  - ou utilizando um objecto do tipo Comparator no construtor de TreeSet. *(vamos ver isto mais tarde)*
- ❖ Implementação baseada numa estrutura em árvore balanceada.
- ❖ Desempenho  $\log(n)$ , para *add*, *remove* e *contains*

# TreeSet – exemplo 1

```
import java.util.TreeSet;

public class Test {
    public static void main(String args[]) {
        TreeSet<String> ts = new TreeSet<>();
        ts.add("viagem");
        ts.add("calendário");
        ts.add("prova");
        ts.add("zircórnio");
        ts.add("ilha do sal");
        ts.add("avião");
        for (String element : ts)
            System.out.println(element + " ");
    }
}
```

```
avião
calendário
ilha do sal
prova
viagem
zircórnio
```

# TreeSet – exemplo 2

```
public class TestTreeSet {
```

```
    public static void main(String[] args) {  
        Collection<Quadrado> c = new TreeSet<>();  
        c.add(new Quadrado(3, 4, 5.6));  
        c.add(new Quadrado(1, 5, 4));  
        c.add(new Quadrado(0, 0, 6));  
        c.add(new Quadrado(4, 6, 7.4));  
        System.out.println(c);
```

```
        for (Quadrado q: c)  
            System.out.println(q);
```

```
    }
```

```
}
```

[Quadrado de Centro (1.0,5.0) e de lado 4.0, Quadrado de Centro (3.0,4.0) e de lado 5.6, Quadrado de Centro (0.0,0.0) e de lado 6.0, Quadrado de Centro (4.0,6.0) e de lado 7.4]

Quadrado de Centro (1.0,5.0) e de lado 4.0

Quadrado de Centro (3.0,4.0) e de lado 5.6

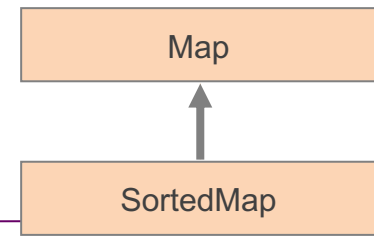
Quadrado de Centro (0.0,0.0) e de lado 6.0

Quadrado de Centro (4.0,6.0) e de lado 7.4

Ordem



# Mapas - Map



- ❖ A Interface *Map* não descende de *Collections*
  - Interface `Map<K,V>`
- ❖ Um mapa é um conjunto que associa uma chave (K) a um valor (V)
  - Não contém chaves duplicadas
- ❖ Também é denominado como dicionário ou memória associativa
- ❖ Métodos disponíveis:
  - adicionar: `put(K key, V value)`
  - remover : `remove(Object key)`
  - obter um objecto: `get(Object key)`

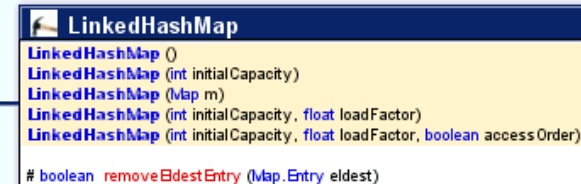
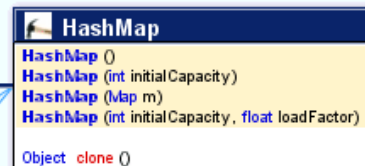
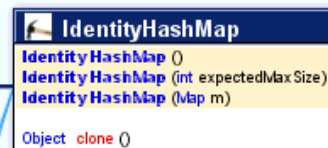
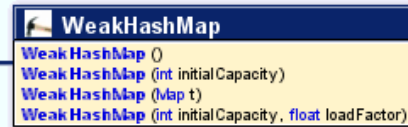
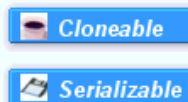
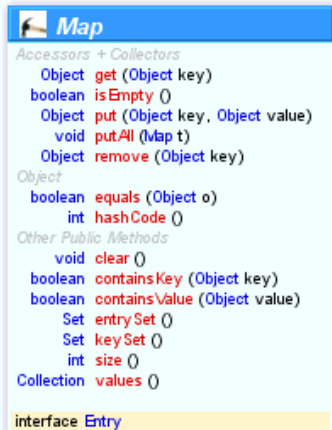
# Classes

java.util.\*

## Map

Methods declared in Interfaces are hidden in subtypes

See also: [Legacy Collection Diagram](#)



www.falkhausen.de Version 0.9 Copyright 2002-04 by Markus Falkhausen. All rights reserved.

# Interface Map<K,V>

```
public interface Map<K,V> {  
    // Basic operations  
    V put(K key, V value);  
    V get(Object key);  
    V remove(Object key);  
    boolean containsKey(Object key);  
    boolean containsValue(Object value);  
    int size();  
    boolean isEmpty();  
    // Bulk operations  
    void putAll(Map<? extends K, ? extends V> m);  
    void clear();  
    // Collection Views  
    public Set<K> keySet();  
    public Collection<V> values();  
    public Set<Map.Entry<K,V>> entrySet();  
    // Interface for entrySet elements  
    public interface Entry {  
        K getKey();  
        V getValue();  
        V setValue(V value);  
    }  
}
```



**Vistas**

# Vistas

---

- ❖ Mapas não são Collections.
- ❖ No entanto, podemos obter vistas dos mapas.
- ❖ As vistas são do tipo Collections
- ❖ Há três vistas disponíveis:
  - conjunto (set) de chaves
  - colecção de valores
  - conjunto (set) de entradas do tipo par chave/valor

# Map – Implementações

---

## ❖ HashMap

- Utiliza uma tabela de dispersão (Hash Table)
- Não existe ordenação nos pares

## ❖ LinkedHashMap

- Semelhante ao HashMap, mas preserva a ordem de inserção

## ❖ TreeMap

- Baseado numa árvore balanceada
- Os pares são ordenados com base na chave
- O desempenho para inserção e remoção é  $O(\log N)$

# HashMap – exemplo

```
public static void main(String[] args) {  
    Map<String, Double> mapa = new HashMap<>();  
    mapa.put("Rui", 32.4);  
    mapa.put("Manuel", 3.2);  
    mapa.put("Rita", 5.6);  
  
    System.out.println("O Mapa contém " + mapa.size() + " elementos");  
    System.out.println("O Rui está no Mapa? " + mapa.containsKey("Rui"));  
  
    System.out.println("A Rita tem " + mapa.get("Rita") + "€");  
    mapa.put("Rita", mapa.get("Rita") + 3.6);  
    System.out.println("A Rita tem " + mapa.get("Rita") + "€");  
  
    Set<Entry<String, Double>> set = mapa.entrySet();  
    for (Entry<String, Double> ele: set)  
        System.out.println("O " + ele.getKey() + " ganha "  
                           + ele.getValue() + "€");  
}
```

O Mapa contém 3 elementos  
O Rui está no Mapa? true  
A Rita tem 5.6€  
A Rita tem 9.2€  
O Manuel ganha 3.2€  
O Rui ganha 32.4€  
O Rita ganha 9.2€

Vista

# TreeMap

---

- ❖ Mesmas características das descritas para a TreeSet mas adaptadas a pares key/value.
- ❖ TreeMap oferece a possibilidade de ordenar objetos
  - utilizando a “Ordem Natural” (compareTo) ou um objeto do tipo Comparator
  - utilização semelhante aos exemplos de HashSet

# Iterar sobre coleções

---

## ❖ Iterator

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();    //optional  
}
```

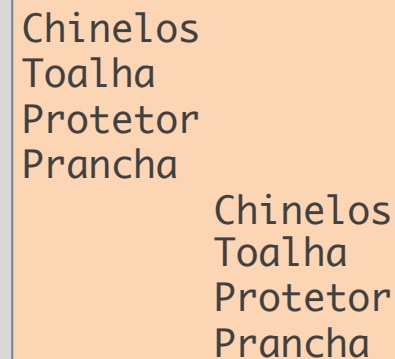
## ❖ ciclo "for each"

```
List<String> names = new LinkedList<>();  
    // ... add some names to the collection  
  
for (String name : names)  
    System.out.println(name);
```



# Exemplo

```
public static void main(String args[]) {  
  
    // vector para simular a entrada de dados  
    String[] acessorios = {"Chinelos", "Toalha", "Protetor", "Prancha"};  
  
    List<String> saco = new ArrayList<>();  
    for (String obj: acessorios )  
        saco.add(obj);  
  
    // Iterador  
    Iterator<String> itr = saco.iterator();  
    while ( itr.hasNext() )  
        System.out.println( itr.next() );  
    // for  
    for (String s: saco)  
        System.out.println("\t"+s );  
}  
}
```



Chinelos  
Toalha  
Protetor  
Prancha

Chinelos  
Toalha  
Protetor  
Prancha

# Exercícios

---

- ❖ Crie estruturas de dados adequadas para conter informação sobre:
  - medidas de temperatura
  - livros (nome e autor)
  - músicas (nome, autor, formato (MP3, WMA, WAV))
  - grupos de 2 elementos numa disciplina
  - agenda de contatos (nome, endereço, cpostal, telefone)
  - ementas (nome, preço) dos restaurantes de Aveiro

# Sumário

---

- ❖ Organização e Principais Interfaces
- ❖ Conjuntos (HashSet e TreeSet)
- ❖ Listas (ArrayList e LinkedList)
- ❖ Mapas (HashMap e TreeMap)