

# Coleções

# JAVA Collections Framework (JCF)

---

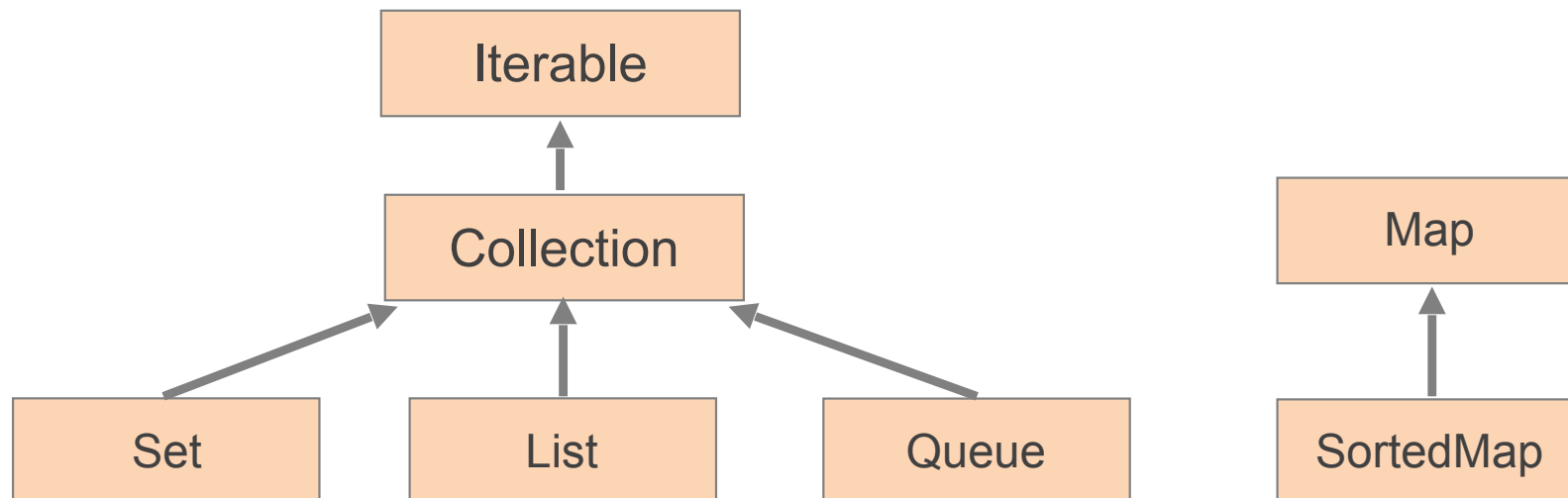
- ❖ Conjunto de classes, interfaces e algoritmos que representam vários tipos de estruturas de armazenamento de dados
  - Listas, Vectors, Pilhas, Árvores, Mapas,...
  - Permitem agregar objetos de determinado tipo paramétrico
  - Exemplo:

```
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, ...*)

# 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

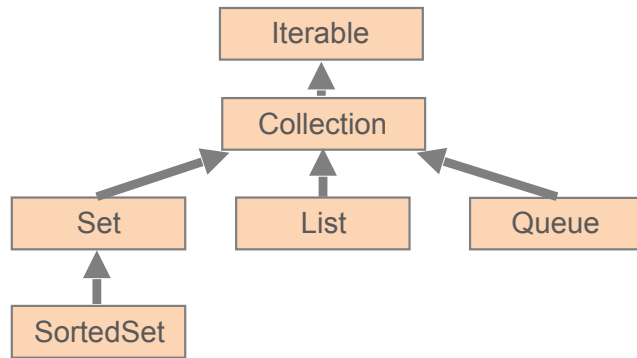
---

- ❖ 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>**

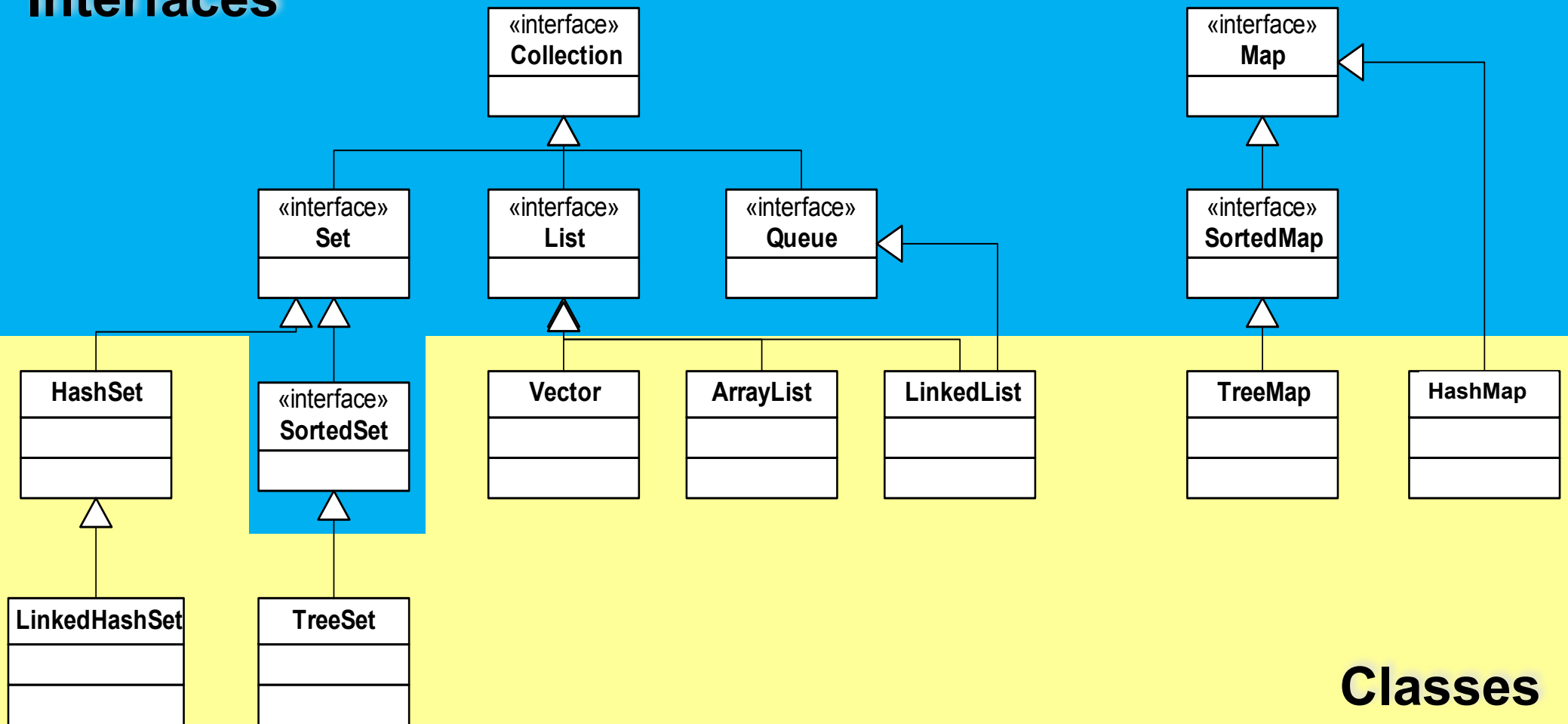
+add(E):boolean  
+remove(Object):boolean  
+contains(Object):boolean  
+size():int  
+iterator():Iterator<E> etc...

<<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



Classes

# Interfaces e Implementações

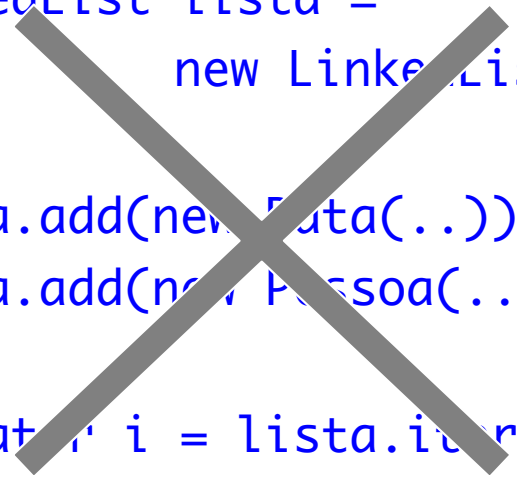
Collections					
	Implementações				
Interfaces	Hash table	Resizable array	Balanced Tree (sorted)	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

# Genéricos em Collections

Desde o JAVA 5 que as Collections são parametrizáveis

Antes..

```
LinkedList lista =  
    new LinkedList();  
  
lista.add(new Data(..));  
lista.add(new Pessoa(..));  
  
Iterator i = lista.iterator();  
  
Data d = (Data)i.next();  
Pessoa p = (Pessoa)i.next();
```



Agora..

```
LinkedList<Data> lista =  
    new LinkedList<Data>();
```

```
lista.add(new Data(..));  
lista.add(new Pessoa(..));
```

**Compile-Time Error**

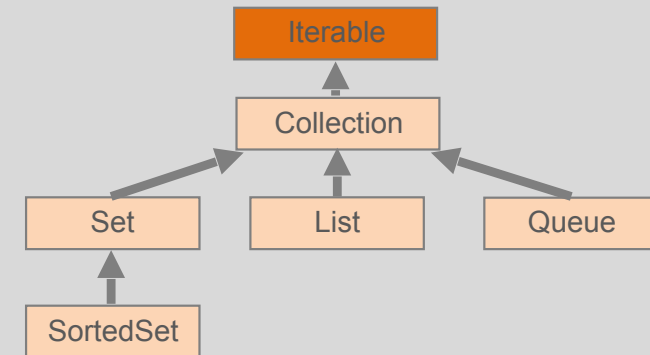
```
Iterator<Data> i =  
    lista.iterator();
```

```
Data d = i.next();  
Pessoa p = (Pessoa)i.next();
```

**Compile-Time Error**



# 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.
```

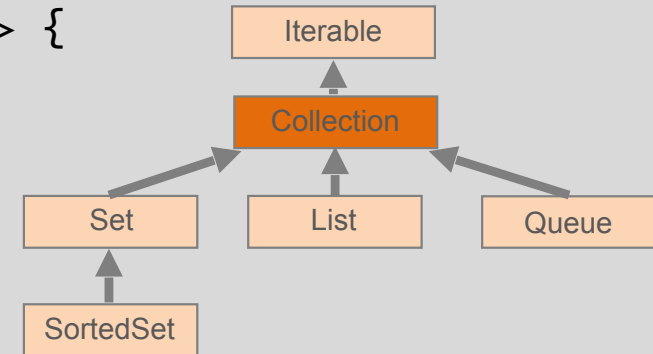
```
    default Spliterator<T> spliterator()
```

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

```
}
```

# Interface Collection

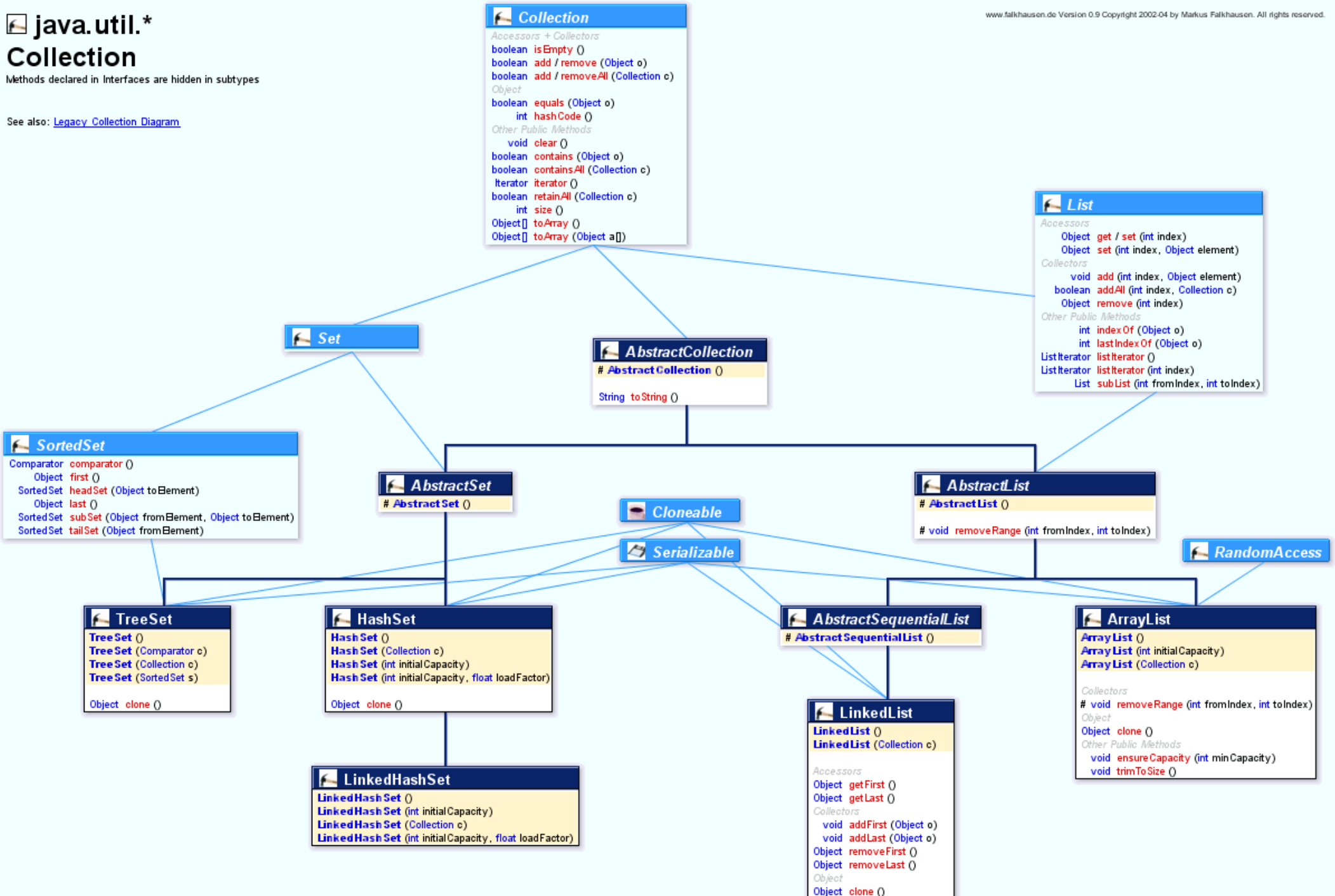
```
public interface Collection<E> extends Iterable<E> {  
    // Basic operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element); //optional  
    boolean remove(Object element); //optional  
    Iterator<E> iterator();  
    // Bulk operations  
    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);  
}
```



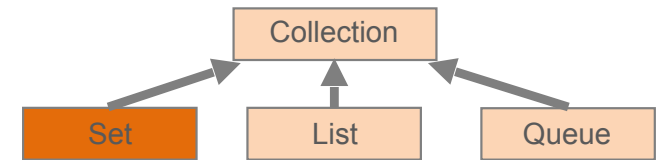
# Collection

Methods declared in Interfaces are hidden in subtypes

See also: [Legacy Collection Diagram](#)



# 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
  - ..

# 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

---

- ❖ A implementação baseada numa estrutura em árvore balanceada.
- ❖ Desempenho  $\log(n)$ , para add, remove e contains
- ❖ 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.



# TreeSet

```
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)
```

```
    }
```

```
}
```

[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

# TreeSet – ordem (Solução 1)

```
class LengthComparator implements Comparator<String> {  
    @Override public int compare(String a, String b) {  
        return (a.length() > b.length() ? 1: -1);  
    }  
}  
  
public class Testes {  
    public static void main(String args[]) {  
        TreeSet<String> ts = new TreeSet<>(new LengthComparator());  
        ts.add("jgdshj");  
        ts.add("hj");  
        ts.add("khsdfk jjskfk");  
        ts.add("f");  
        ts.add("opeiwoj kn kndsjsa");  
        ts.add("kndkd");  
        for (String element : ts)  
            System.out.println(element + " ");  
    }  
}
```

```
f  
hj  
kndkd  
jgdshj  
khsdfk jjskfk  
opeiwoj kn kndsjsa
```

# TreeSet – ordem (Solução 2)

```
public class Testes {  
    public static void main(String args[]) {  
        TreeSet<String> ts = new TreeSet<>(new Comparator<String>() {  
            @Override public int compare(String a, String b) {  
                return (a.length() > b.length() ? 1 : -1);  
            }  
        });  
  
        ts.add("jgdshj");  
        ts.add("hj");  
        ts.add("khsdfk jjskfk");  
        ts.add("f");  
        ts.add("opeiwoj kn kndsjsa");  
        ts.add("kndkd");  
  
        for (String element : ts)  
            System.out.println(element + " ");  
    }  
}
```

f  
hj  
kndkd  
jgdshj  
khsdfk jjskfk  
opeiwoj kn kndsjsa

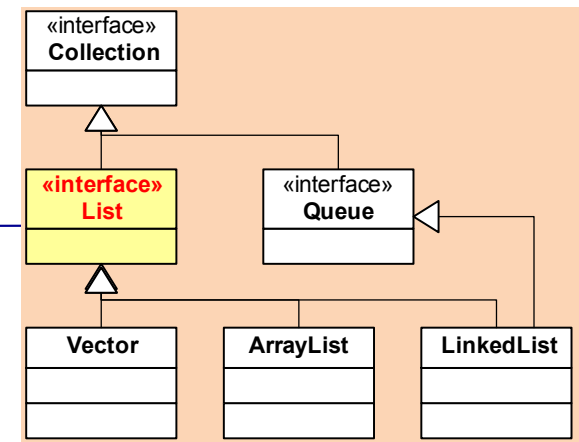
# TreeSet – ordem (Solução 3 – Java 8)

```
public class Testes {  
    public static void main(String args[]) {  
        TreeSet<String> ts =  
            new TreeSet<>(Comparator.comparing(String::length));  
  
        ts.add("jgdshj");  
        ts.add("hj");  
        ts.add("khsdfk jjskfk");  
        ts.add("f");  
        ts.add("opeiwoj kn kndsjsa");  
        ts.add("kndkd");  
  
        for (String element : ts)  
            System.out.println(element + " ");  
    }  
}
```

Método referência (Java 8)

```
f  
hj  
kndkd  
jgdshj  
khsdfk jjskfk  
opeiwoj kn kndsjsa
```

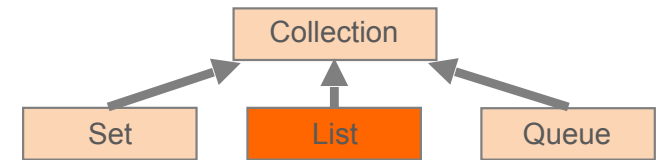
# Listas



- ❖ 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);  
    E get(int index);  
    E set(int index, E element);  
    E remove(int index);  
    boolean addAll(Collection<? extends E> c);  
  
    // 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);  
}
```

// Optional

// Optional

// Optional

// Optional

```
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?

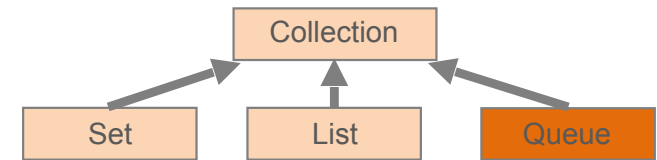
# Listas – Exemplo

```
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



# Queue – Filas



```
public interface Queue<E> extends Collection<E> {  
  
    // Inserts the specified element in the queue  
    // throws an exception if empty  
    boolean add(E 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();  
}
```

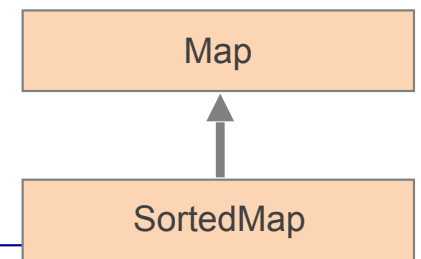


# Filas - Implementações

---

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

# 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](#)

**Map**

Accessors + Collectors

Object

get (Object key)

boolean

isEmpty ()

Object

put (Object key, Object value)

void

putAll (Map t)

Object

remove (Object key)

Object

boolean

equals (Object o)

int

hashCode ()

Other Public Methods

void

clear ()

boolean

containsKey (Object key)

boolean

containsValue (Object value)

Set

entrySet ()

Set

keySet ()

int

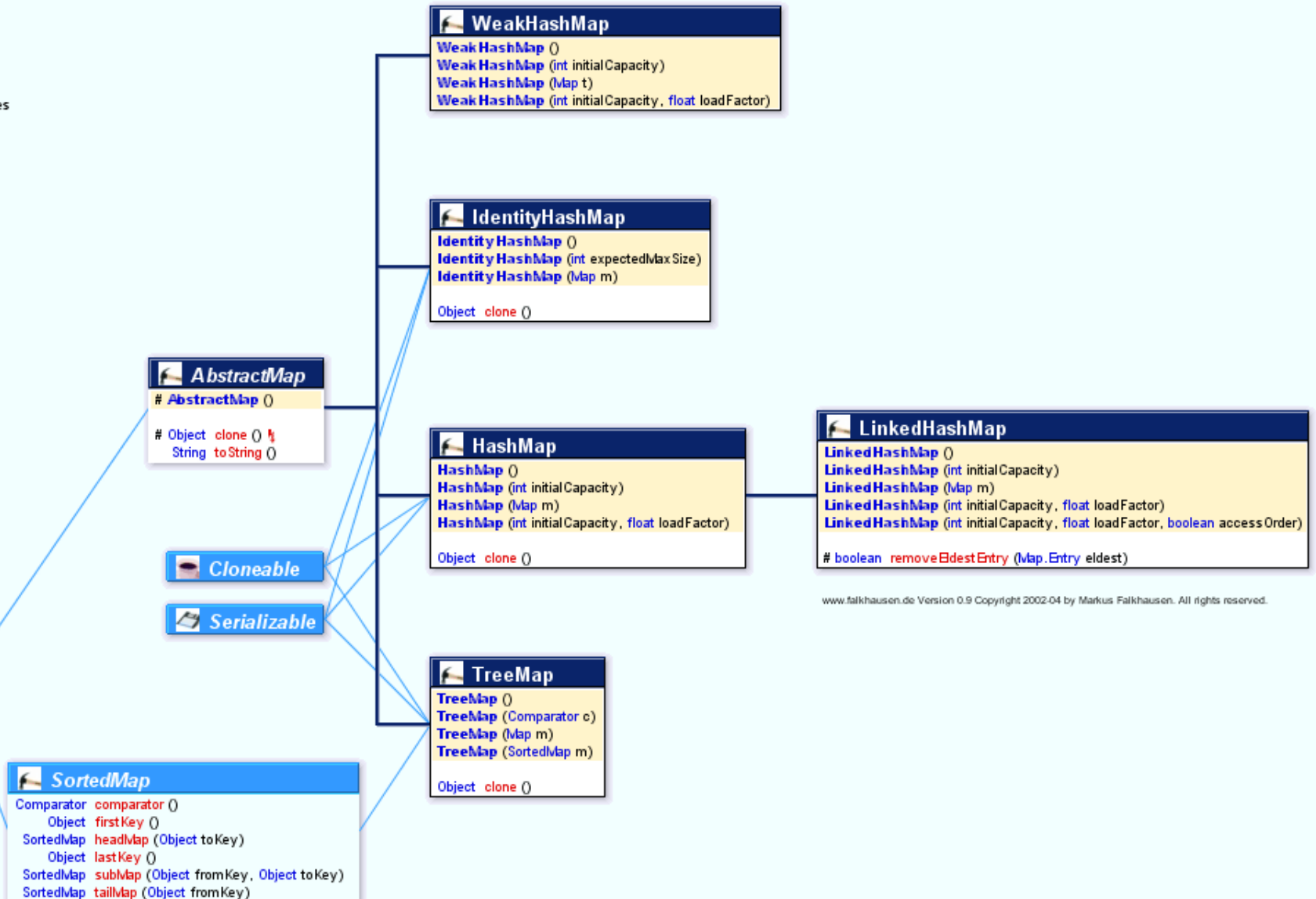
size ()

Collection

values ()

interface

Entry



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);
```

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

```
    System.out.println("0 Mapa contém " + mapa.size() + " elementos");  
    System.out.println("0 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("0 " + ele.getKey() + " ganha "  
                            + ele.getValue() + "€");
```

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);
```

# Iterar sobre coleções (java 8)

---

## ❖ método forEach

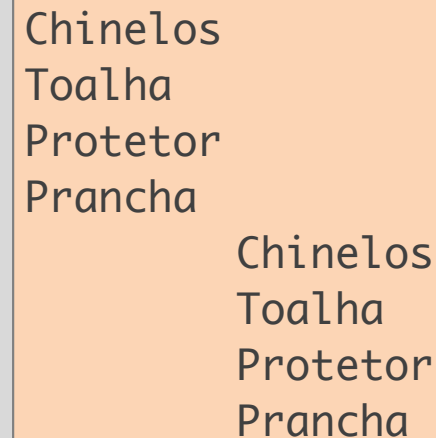
```
List<String> names = new LinkedList<>();  
    // ... add some names to the collection  
  
names.forEach(name -> System.out.println(name));
```

## ❖ streams

```
List<String> names = new LinkedList<>();  
    // ... count some names in the collection  
  
long count = names.stream()  
    .filter(name -> name.startsWith("A"))  
    .count();
```

# Exemplos

```
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
```

# Exemplos (java 8)

```
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);  
  
    // forEach  
    saco.forEach(name -> System.out.println(name));  
    // stream  
    long count = saco.stream()  
        .filter(name -> name.startsWith("P"))  
        .count();  
    System.out.println( count );  
}  
}
```

Chinelos  
Toalha  
Protetor  
Prancha  
2

# Algoritmos

---

- ❖ A JCF fornece ainda um conjunto de algoritmos que podem ser usados em coleções
- ❖ Métodos estáticos de utilização global
  - `java.util.Collections`
  - `java.util.Arrays`
- ❖ Exemplos:
  - `sort`, `binarySearch`, `copy`, `shuffle`, `reverse`, `max`, `min`, etc.

# java.util.Collections

---

❖ Note a diferença!!

❖ Classe

– **java.util.Collections**

❖ Interface

– java.util.Collection

static <T> boolean	<b>addAll</b> ( <b>Collection</b> <? super T> c, T... elements)	Adds all of the specified elements to the specified collection.
static <T> int	<b>binarySearch</b> ( <b>List</b> <? extends <b>Comparable</b> <? super T>> list, T key)	Searches the specified list for the specified object using the binary search algorithm.
static <E> <b>Collection</b> <E>	<b>checkedCollection</b> ( <b>Collection</b> <E> c, <b>Class</b> <E> type)	Returns a dynamically typesafe view of the specified collection.
static <T> void	<b>copy</b> ( <b>List</b> <? super T> dest, <b>List</b> <? extends T> src)	Copies all of the elements from one list into another.
static boolean	<b>disjoint</b> ( <b>Collection</b> <?> c1, <b>Collection</b> <?> c2)	Returns true if the two specified collections have no elements in common.
static <T> <b>List</b> <T>	<b>emptyList</b> ()	Returns the empty list (immutable).
static <T> void	<b>fill</b> ( <b>List</b> <? super T> list, T obj)	Replaces all of the elements of the specified list with the specified element.
static int	<b>frequency</b> ( <b>Collection</b> <?> c, <b>Object</b> o)	Returns the number of elements in the specified collection equal to the specified object.
static int	<b>indexOfSubList</b> ( <b>List</b> <?> source, <b>List</b> <?> target)	Returns the starting position of the first occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
static int	<b>lastIndexOfSubList</b> ( <b>List</b> <?> source, <b>List</b> <?> target)	Returns the starting position of the last occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
static <T> T	<b>max</b> ( <b>Collection</b> <? extends T> coll, <b>Comparator</b> <? super T> comp)	Returns the maximum element of the given collection, according to the order induced by the specified comparator.
static <T> T	<b>min</b> ( <b>Collection</b> <? extends T> coll, <b>Comparator</b> <? super T> comp)	Returns the minimum element of the given collection, according to the order induced by the specified comparator.
static <T> boolean	<b>replaceAll</b> ( <b>List</b> <T> list, T oldVal, T newVal)	Replaces all occurrences of one specified value in a list with another.
static void	<b>reverse</b> ( <b>List</b> <?> list)	Reverses the order of the elements in the specified list.
static void	<b>rotate</b> ( <b>List</b> <?> list, int distance)	Rotates the elements in the specified list by the specified distance.
static void	<b>shuffle</b> ( <b>List</b> <?> list)	Randomly permutes the specified list using a default source of randomness.
static <T> void	<b>sort</b> ( <b>List</b> <T> list, <b>Comparator</b> <? super T> c)	Sorts the specified list according to the order induced by the specified comparator.
static <T> <b>Collection</b> <T>	<b>unmodifiableCollection</b> ( <b>Collection</b> <? extends T> c)	Returns an unmodifiable view of the specified collection.



# Ordenação – exemplo (ordem natural)

```
public static void main(String[] args) {  
    List<Integer> list = new ArrayList<>();  
  
    for (int i=0;i<10;i++) {  
        list.add((int) (Math.random() * 100));  
    }  
  
    System.out.println("Initial List: "+list);  
    Collections.sort(list);  
    System.out.println("Sorted List:  "+list);  
}
```

Initial List: [57, 27, 83, 4, 73, 34, 74, 74, 0, 46]  
Sorted List: [0, 4, 27, 34, 46, 57, 73, 74, 74, 83]

# Ordenação – exemplo (comparator)

```
public static void main(String[] args) {
    ArrayList<Integer> randInts = new ArrayList<>();
    // generate 5 random ints for randInts
    Random rnd = new Random();
    for (int i=0;i<5;i++)
        randInts.add(rnd.nextInt());

    System.out.println(randInts);

    // sort the randInts ArrayList
    Collections.sort(randInts, new Comparator<Integer>() {
    @Override public int compare(Integer o1, Integer o2) {
        return (o2.intValue() > o1.intValue()) ? 1 : -1;
    }
    });
    System.out.println([ 173157445 -158699807 -883988021 -1720841220 285295582 ]
                       [ 285295582 173157445 -158699807 -883988021 -1720841220 ]
    )
}
```

# Ordenação - exemplo (java 8)

```
public static void main(String[] args) {
    System.out.println("--Sorting with natural order");
    List<String> l1 = createList();
    l1.sort(null);
    l1.forEach(System.out::println);
    System.out.println("--Sorting with a lambda expression");
    List<String> l2 = createList();
    l2.sort((s1, s2) -> s1.compareToIgnoreCase(s2)); // sort ignoring case
    l2.forEach(System.out::println);
    System.out.println("--Sorting with a method reference");
    List<String> l3 = createList();
    l3.sort(String::compareToIgnoreCase);
    l3.forEach(System.out::println);
}

private static List<String> createList() {
    List<String> list = new ArrayList<>();
    list.add("Ubuntu");
    list.add("Android");
    list.add("Mac OS X");
    return list;
}
```

```
--Sorting with natural order
Android
Mac OS X
Ubuntu
--Sorting with a lambda expression
Android
Mac OS X
Ubuntu
--Sorting with a method reference
Android
Mac OS X
Ubuntu
```

# java.util.Arrays

## Arrays

+asList(a: Object[]) : List  
+binarySearch(a: byte[], key: byte) : int  
+binarySearch(a: char[], key: char) : int  
+binarySearch(a: double[], key: double) : int  
+binarySearch(a: float[], key: float) : int  
+binarySearch(a: int[], key: int) : int  
+binarySearch(a: long[], key: long) : int  
+binarySearch(a: Object[], key: Object) : int  
+binarySearch(a: Object[], key: Object, c: Comparator) : int  
+binarySearch(a: short[], key: short) : int  
+equals(a: boolean[], a2: boolean[]) : boolean  
+equals(a: byte[], a2: byte[]) : boolean  
+equals(a: char[], a2: char[]) : boolean  
+equals(a: double[], a2: double[]) : boolean  
+equals(a: float[], a2: float[]) : boolean  
+equals(a: int[], a2: int[]) : boolean  
+equals(a: long[], a2: long[]) : boolean  
+equals(a: Object[], a2: Object[]) : boolean  
+equals(a: short[], a2: short[]) : boolean  
+fill(a: boolean[], val: boolean) : void  
+fill(a: boolean[], fromIndex: int, toIndex: int, val: boolean) : void

Overloaded fill method for char, byte, short, int, long, float, double, and Object.

+sort(a: byte[]) : void  
+sort(a: byte[], fromIndex: int, toIndex: int) : void

Overloaded sort method for char, short, int, long, float, double, and Object.

# 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 contactos (nome, endereço, cpostal, telefone)
  - ementas (nome, preço) dos restaurantes de Aveiro

# JAVA Collections Framework – sumário

---

- ❖ Organização e Principais Interfaces
  - ❖ Conjuntos (HashSet e TreeSet)
  - ❖ Listas (ArrayList e LinkedList)
  - ❖ Mapas (HashMap e TreeMap)
  - ❖ Operações sobre Coleções
- 
- ❖ Java 8 introduzir muitas alterações !!