Java Orientado a Objetos
Collections

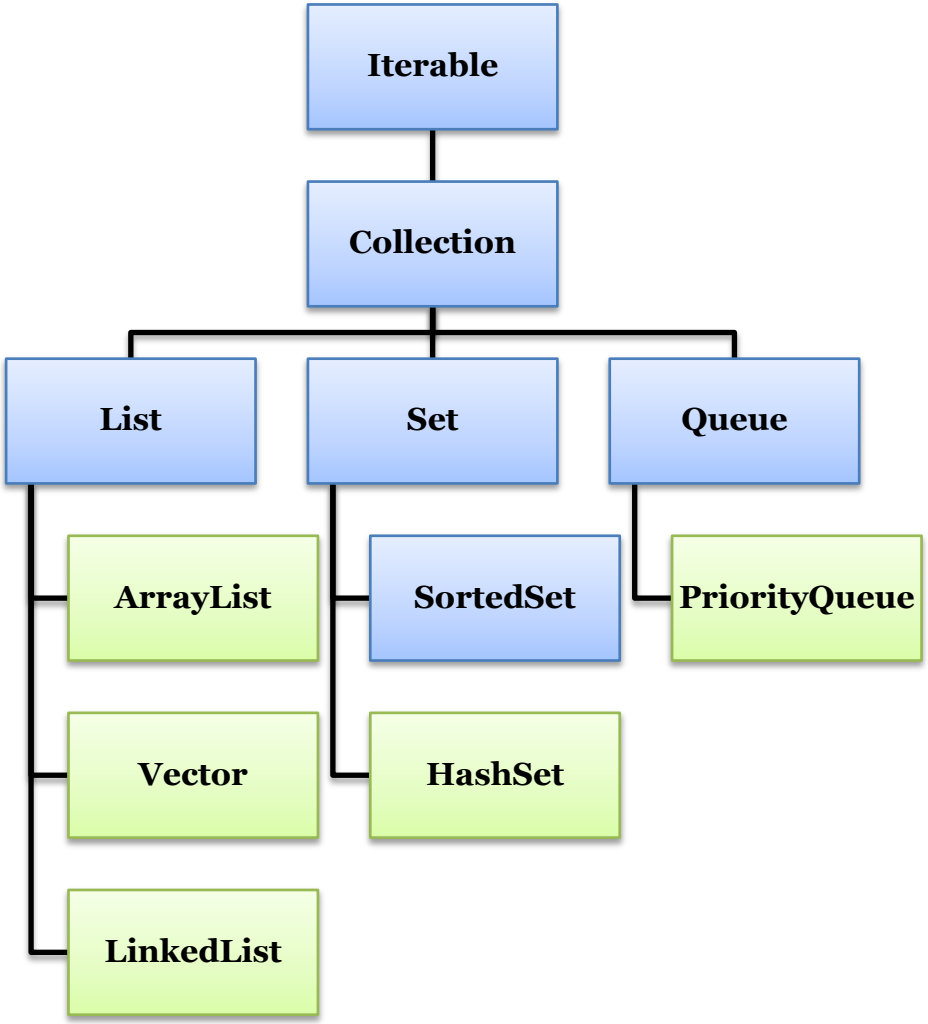
3WAY NETWORKS

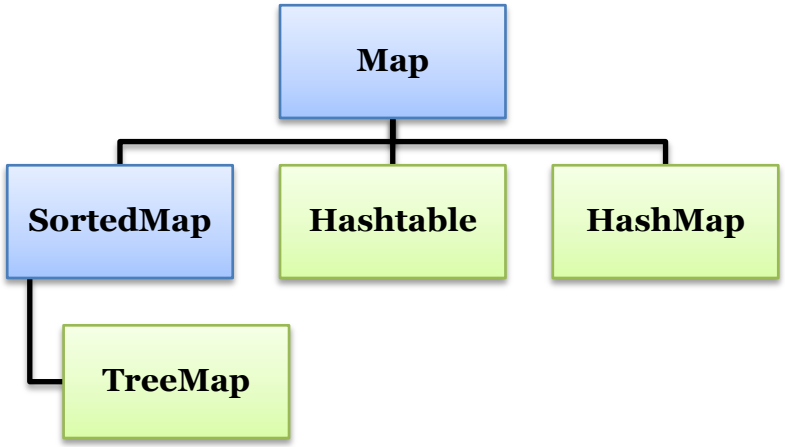
Java Collections

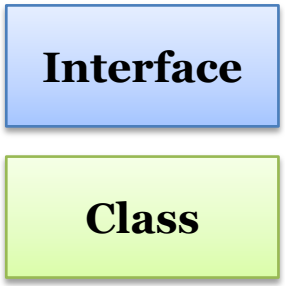
É um objeto onde podemos agrupar vários elementos (outros objetos)

Métodos implementados que realizam operações(sort, reverse, isEmpty, size ...) sobre as coleções

Java Collections - Hierarquia





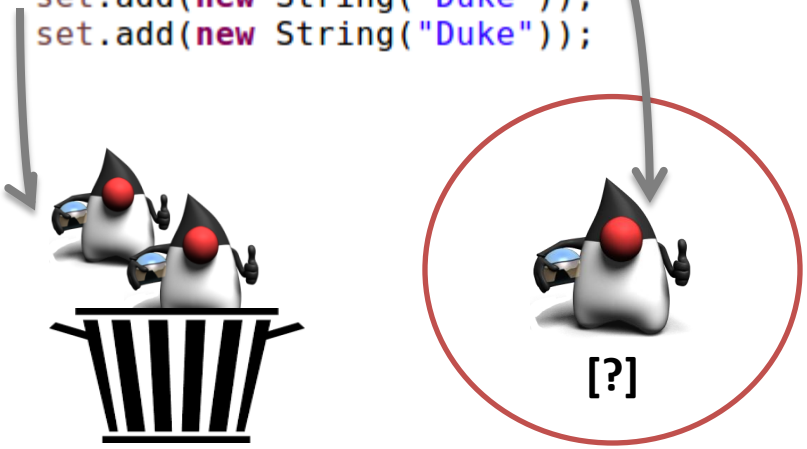


Interfaces Set e List

Interface Set

```
Set<String> set = new HashSet<>();

set.add(new String("Duke"));
set.add(new String("Duke"));
set.add(new String("Duke"));
```

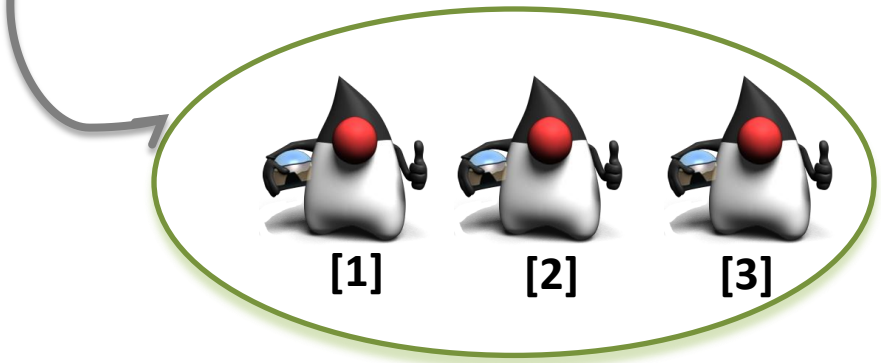


Coleções não ordenadas que não contém duplicidades

Interface List

```
List<String> list = new ArrayList<>();

list.add(new String("Duke"));
list.add(new String("Duke"));
list.add(new String("Duke"));
```

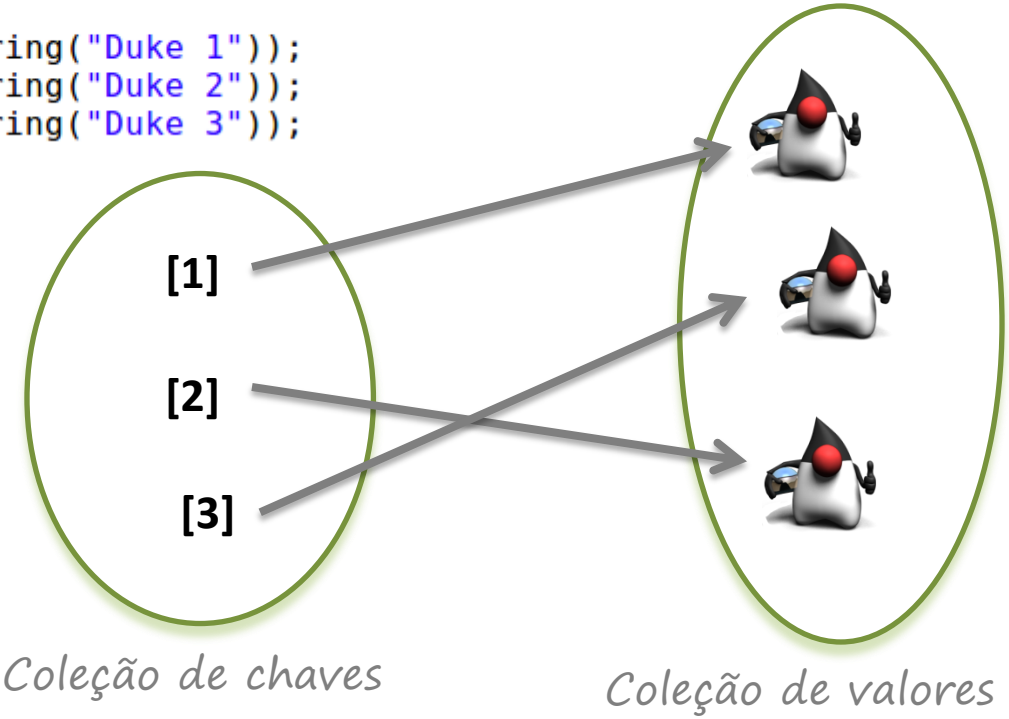


Coleções de classes ordenadas onde as duplicidades são permitidas.

Interface Map

```
Map<Integer,Object> map = new HashMap<>();

map.put(1, new String("Duke 1"));
map.put(2, new String("Duke 2"));
map.put(3, new String("Duke 3"));
```



Use map.get([chave]) para recuperar objetos

Generics e Coleções Java

Métodos de **java.util.Collection<E>:**

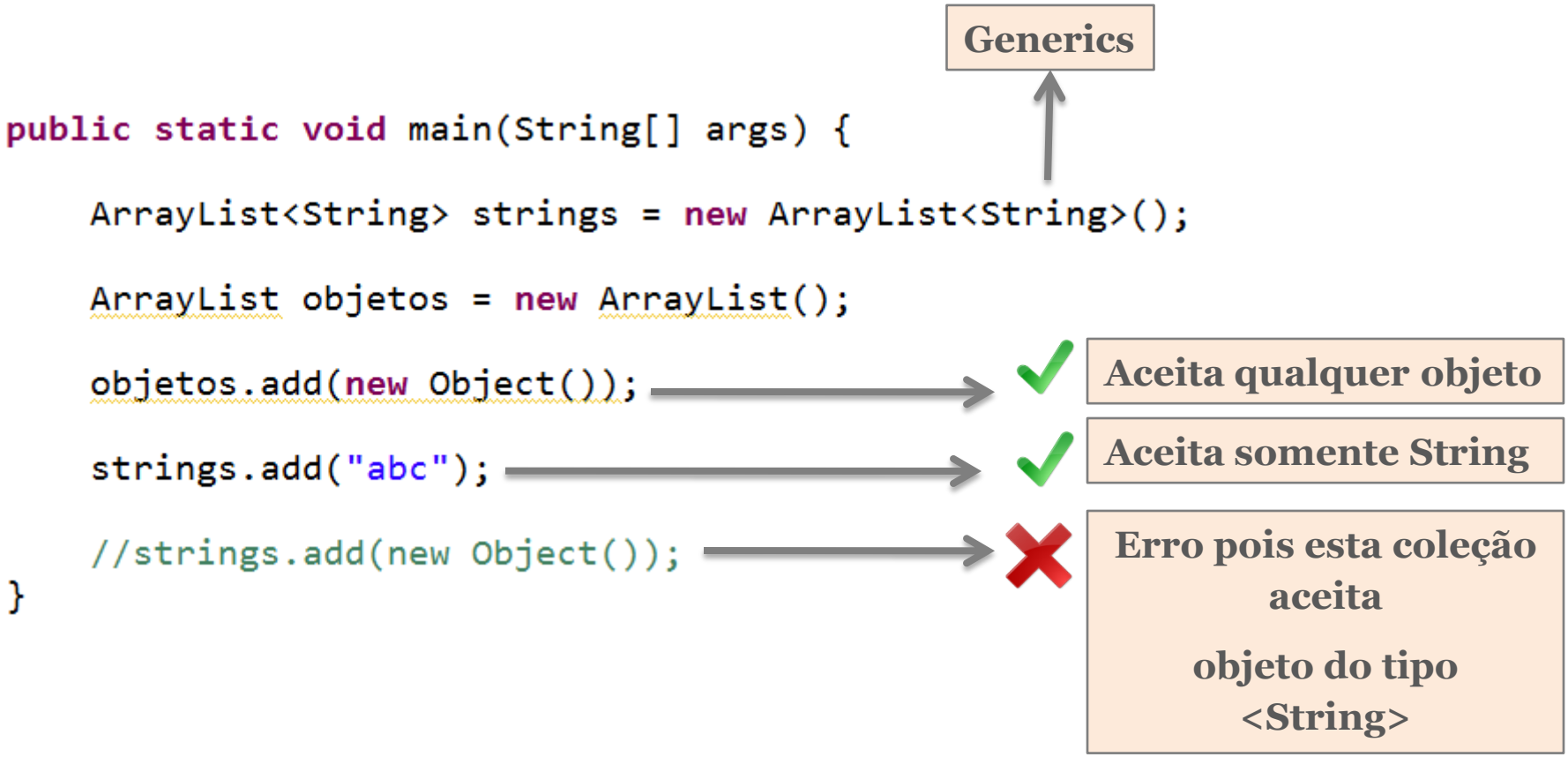
```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
boolean contains(Object o)
boolean containsAll(Collection<?> c)
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
void clear()
int size()
boolean isEmpty()
Object[] toArray()
<T> T[] toArray(T[] a)
```

Generics

<?> coringa

```
//JAVA 8
default Spliterator<E> spliterator()
default Stream<E> stream()
default Stream<E> parallelStream()
```

Generics e Coleções Java



Interface Iterator, Iterable

Uma referência **Iterator** é obtido na própria coleção, que **implementa Iterable**, através do método **iterator()**

```
public interface Iterable<T> {
  Iterator<T> iterator();
  default void forEach(Consumer<? super T> action)
  default Spliterator<T> spliterator()
}
```

A partir de Java 8

```
public interface Iterator<E> {
      //retorna true se houver mais elementos a iterar
      boolean hasNext();
      //retorna o proximo elemento na proxima iteracao
      E next();
   //remove o ultimo elemento retornado pela iteracao
      void remove();
}
```

Percorrendo Collections

```
ArrayList<String> strings = new ArrayList<String>();
strings.add("String1");
strings.add("String2");
strings.add("String3");
```

Enhaced-for	Iterator

`for (String str : strings) {`
`    System.out.println(str);`
`}`
	`Iterator<String> iterator = strings.iterator();`
	`while (iterator.hasNext()) {`
	`    System.out.println(iterator.next());`
	`}`

Java 8 – método forEach

```
ArrayList<String> strings = new ArrayList<String>();
strings.add("String1");
strings.add("String2");
strings.add("String3");
```

Usando forEach de forma imperativa com anonymous inner class

```
strings.forEach(new Consumer<String>() {
        @Override
        public void accept(final String str) {
                System.out.println(str);
        }
});
```

Usando forEach com Lambada

```
strings.forEach((str) -> System.out.println(str));
```

```
strings.forEach(System.out::println);
```

Usando forEach com Lambda, da forma mais reduzida, com métodos de referência

Classificando Coleções: *Collections.sort*

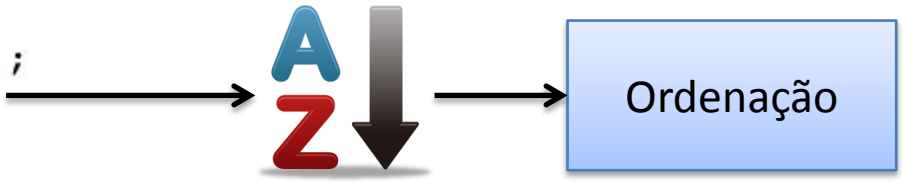
A classe **Collections** nos permite ordenar coleções, através do um método estático **sort**, que recebe um **List** como argumento.

```
import java.util.*;

public class OrdenandoCollection {
      public static void main(String[] args) {
            List<String> lista = new ArrayList<String>();

            lista.add("Goiânia");
            lista.add("São Paulo");
            lista.add("Aracajú");
            // lista sem ordenação
            System.out.println(lista);
            Collections.sort(lista);
            // lista ordenada
            System.out.println(lista);

      }
}
```



Interface Comparable

Para ordenar objetos criados devemos implementar a interface **java.lang.Comparable**, definindo o método **int compareTo(Object)** do objeto criado.

-1 **0** **1**

< **=** **>**

Java 8 – Collections e Streams

A classe **java.util.Stream**, representa uma seqüência de elementos. **Streams** são como "dutos" jorrando objetos, você pode aplicar uma ou mais operações

Prepara um "duto" de objetos e repassa para função intermediária sorted()

Pega objetos do "duto" e invoca compareTo(), repassa para função filter()

```
colecaoDeObjetos
        .stream()

        .sorted()

        .filter((o) -> o.getNome().startsWith("m"))

        .forEach(System.out::println);
```

Usa método de referência para imprimir toString() dos objetos que satisfazeram o filtro

Pega objetos do "duto" e invoca expressão lambida para filtrar objetos com campo nome começando com caractere "m", repassa objetos para função forEach()