Programação Orientada por Objectos

Introdução à Coleções - Lists

Prof. Rui César das Neves, Prof. José Cordeiro

Departamento de Sistemas e Informática

Escola Superior de Tecnologia de Setúbal – Instituto Politécnico de Setúbal

2014/2015

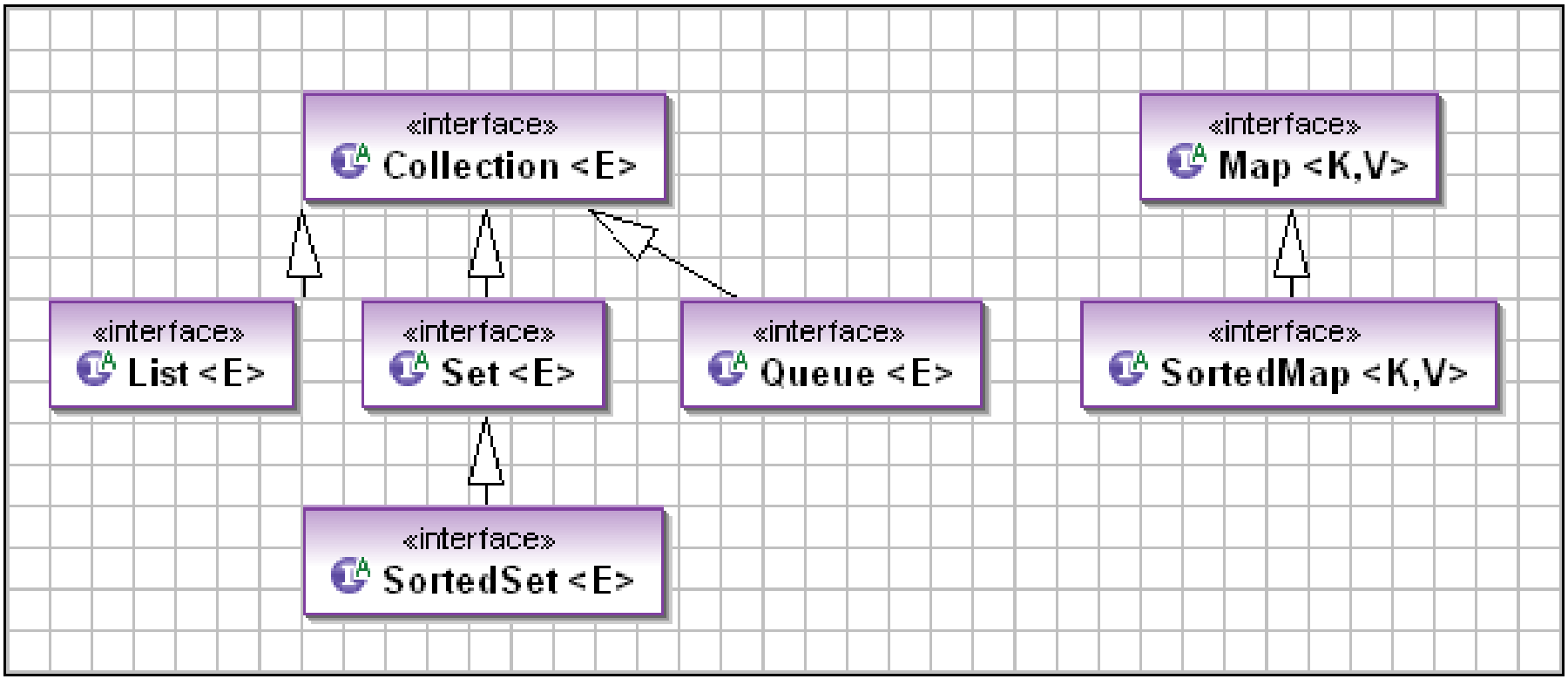
Sumário

- Coleção:
  - O que é?
- Java Collections Framework (JCF)
  - O que é?
  - Interfaces genéricas e Algoritmos Polimórficos
- Listas
  - Interface **`List<E>`**
    - Classe **`ArrayList<E>`**
    - Classe **`LinkedList<E>`**
- Iteração e Alterações em Coleções
    - Ciclo **`For-Each`**
    - Não modificar a coleção durante a iteração
    - Interface e Classe **`Iterator<E>`**

Coleções e a Java Collections Framework

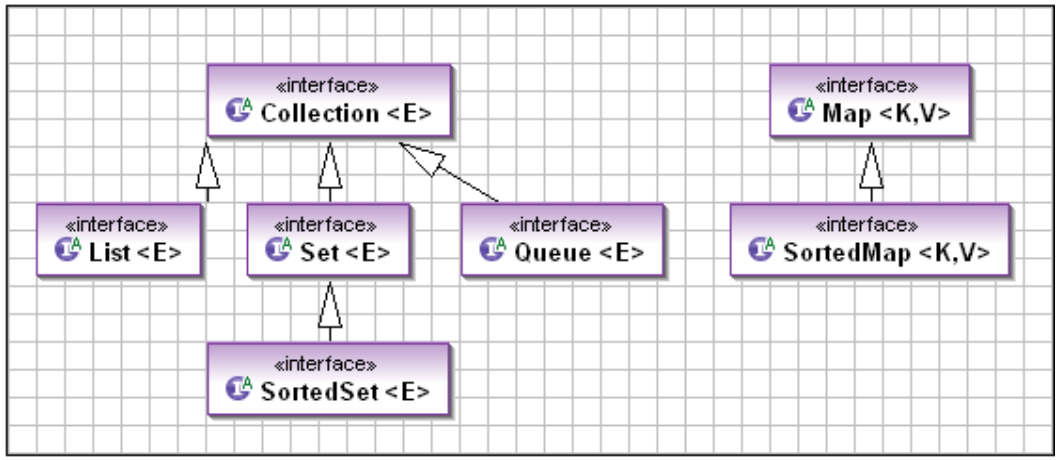
- **Coleções:** agrupam vários elementos numa estrutura de dados organizada, com propriedades e funcionalidades próprias
  - Exemplo: é o caso de uma Fila de espera, uma estrutura linear, em que os elementos que a compõem se arrumam sequencialmente e em que cada novo elemento é inserido no fim da fila e o primeiro elemento a remover é o que primeiro foi inserido. (*FIFO – first in first out*)
- Java Collections Framework (**JCF**):
  - É uma arquitetura unificada que engloba interfaces, classes (abstratas e concretas) e algoritmos (implementados por) métodos capazes de manipular Coleções.
  - A JCF inclui as interfaces de quatro principais tipos de coleções:
    - Conjuntos (Set): Coleção de elementos sem ordem e sem elementos repetidos
    - Listas (List): Coleção de elementos ordenados e com possíveis repetições
    - Mapas (Map): Coleção de pares chave-valor, sem repetição da chave
    - Filas (Queue) : Sequências de elementos com diferentes critérios de inserção e remoção

Java Collections Framework – Interfaces Genéricas



Java Collections Framework – Interfaces Genéricas

- As interfaces definidas na JCF são ditas interfaces genéricas porque as classes que as implementam podem armazenar um qualquer tipo de dados.
  - Note que o tipo de dados a armazenar por essas classes é especificado na sua definição tornando-as passíveis de guardar apenas dados desse tipo.
  - Exemplos:
    - **`public interface Collection<E> …`**
      - **Leia-se:** Colecção de elementos do tipo **'E'**lement
      - A referência sintáctica **<E>** assinala que a interface é genérica. **'E'** pode ser **`String`**, **`Integer`**, ou qualquer referência a outro tipo de objetos.
    - **`public interface Map<K,V> …`**
      - **Leia-se:** mapa de elementos do tipo **'V'**alue com chave do tipo **'K'**ey.

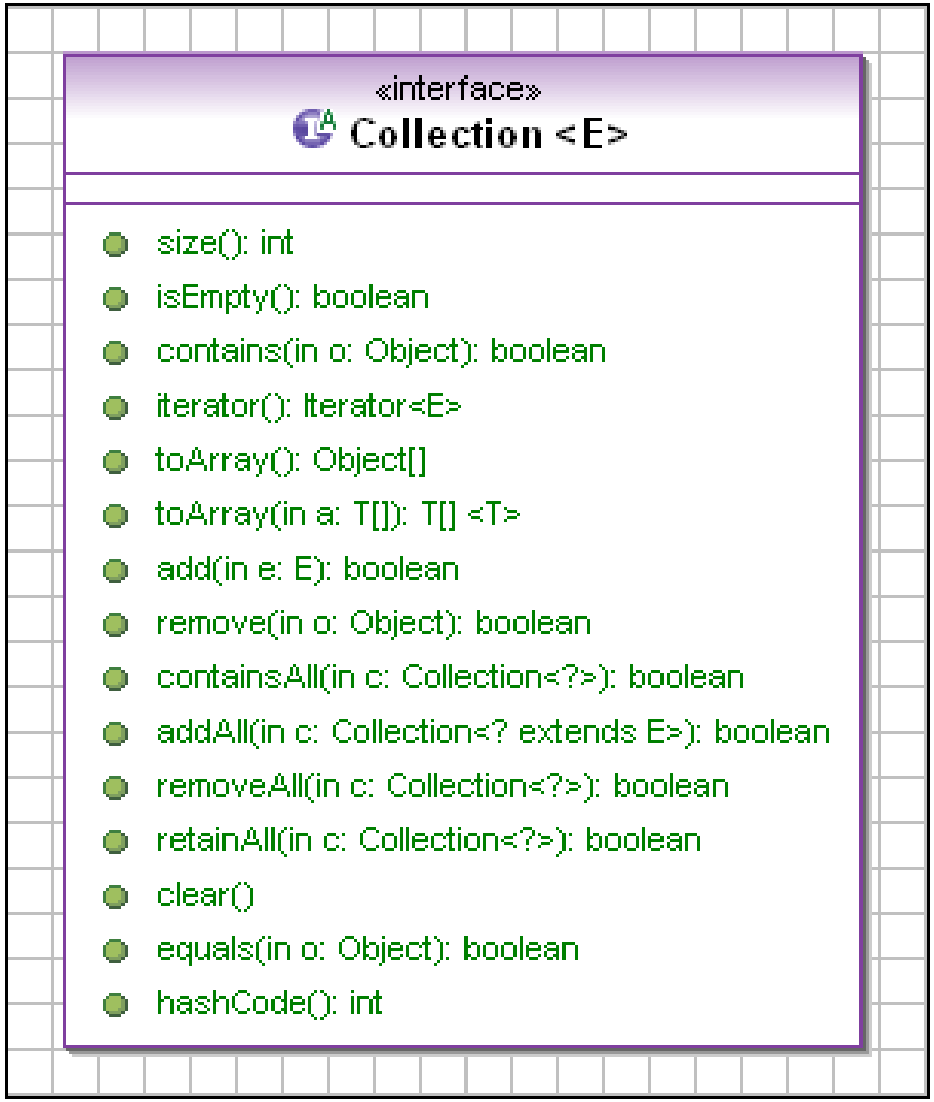


Java Collections Framework – Algoritmos Polimórficos

- A maioria dos algoritmos polimórficos disponibilizados pela JCF através da classe **`Collections`** aplicam-se especificamente a *listas*, alguns deles são:
  - **`sort`** – ordena uma lista por um critério
  - **`shuffle`** – baralha os elementos da lista aleatoriamente
  - **`reverse`** – reverte a ordem dos elementos na lista
  - **`rotate`** – roda todos os elementos da lista numa distância especificada
  - **`swap`** – troca os elementos em posições especificadas de uma lista
  - **`replaceAll`** – troca todas as ocorrências de um valor por um outro valor especificado
  - **`fill`** – atribui a todos os elementos da lista um valor especificado
  - **`copy`** – copia uma lista para outra
  - **`binarySearch`** – procura um elemento numa lista com o algoritmo de procura binária
- Exemplo de evocação de um algoritmo polimórfico:
  - `Collections.rotate(nomeDaLista, 4);`
    - Passa os 4 últimos elementos de **`nomeDalista`** para o principio de **`nomeDalista`** pela mesma ordem em que estavam no final da lista.

Coleções – Interface Collection<E>

- Interface na raiz da hierarquia.
- **`Collection<E>`** é, por isso mesmo, o menor denominador comum de todas as implementações de coleções,
- É útil como referência genérica de coleções, e na manipulação com a máxima generalidade.



Coleções – Interface List<E>

- Interface que impõe uma ordem na coleção.
- As listas podem ter elementos duplicados.
- O cliente de uma Lista tem, normalmente, controlo efetivo sobre o ponto onde um elemento é inserido.
- O acesso a um elemento pode ser feito por um índice (referência de posição).

Coleções – ArrayList<E>

- **ArrayList<E> –** é a implementação da interface **`List<E>`** mais comum com bom desempenho.
  - É a única que estudaremos detalhadamente.
  - **`ArrayList<Pessoa> pessoas = new ArrayList<>();`**
    - Declara e cria uma **`ArrayList`** chamada pessoas para armazenar objetos da classe **`Pessoa`**
- É a implementação fornecida pelo JCF da classe **`Lista<E>`** definida na última aula (da forma mais otimizada possível, não necessariamente com *arrays*)

Coleções – ArrayList<E>

- Exemplo de utilização:

```
public static void main(String[] args) {
  ArrayList<String> lista = new ArrayList<>(); // cria a ArrayList de Strings
  lista.add("IPOO");                           // adiciona a String "IPOO"
  lista.add("POO");                            // adiciona a String "POO" a seguir a "IP"
  lista.add("POO");                            // adiciona a String "POO" a seguir a "POO"
  lista.add("IPOO");                           // adiciona a String "IPOO" a seguir a "POO"
  for(int i = 0; i <lista.size(); ++i) {// itera a lista de [0] a lista.size …
    if (lista.get(i).equals("POO")) {    // se conteúdo de lista.get(i) for igual a "POO"
      lista.remove(i);                   // remove o elemento [i] da lista
    }
  }
  lista.remove("IPOO");                  // remove o 1º elemento "IPOO" da lista
  for(int i = 0; i <lista.size(); ++i) {
    System.out.println(lista.get(i));    // O resultado é surpreendente?
  }
}
```

Coleções – LinkedList<E>

- **LinkedList<E>** – Nas operações de inserção e remoção de elementos pode oferecer melhor desempenho.
  - `LinkedList<Pessoa> pessoas = new LinkedList<>();`
    - Declara e cria uma **`LinkedList`** chamada pessoas para armazenar objetos da classe Pessoa
- Internamente a implementação não utiliza **`arrays`**, para não "desperdiçar" espaço, mas regista, para cada elemento qual o seu próximo e anterior, sendo fácil percorrer a lista e inserir e/ou remover elementos

Coleções – LinkedList<E>

- Exemplo de utilização:

```
public static void main(String[] args) {
  LinkedList<Integer> lista = new LinkedList<>(); //cria LinkedList de int
  for (int val = 0; val < 10; val++) {          //cria valores para val
     lista.add(new Integer(val));               //insere val no fim da lista
  }
  for (int i=0; i< lista.size(); i++) {         //itera, por posição, a LinkedList
     if (lista.get(i).intValue()==5) {          //se conteúdo do elemento [i] == 5
        lista.remove(i);                        //remove o elemento [i]
     }
  }
  for (int i=0; i< lista.size(); i++) {
     System.out.println(lista.get(i));
  }
}
```

Iteração de Coleções - Ciclo For-Each

- Ciclo compacto que pode ser usado para iterar (percorrer sequencialmente) qualquer coleção ou *array*.
  - Este ciclo usa implicitamente um iterador para os objetos do tipo guardado na coleção (no exemplo notar o uso de **`List<String> listaStrings`** em oposição à declaração **`ArrayList<String> listaStrings`**):

```
public static void main(String[] args) {
  List<String> listaStrings = new ArrayList<String>();
  listaStrings.add("AO");
  listaStrings.add("IPOO");
  listaStrings.add("POO");
  listaStrings.add("PV");
  listaStrings.add("POO");
  listaStrings.add("POO");
  for (String ls: listaStrings){    // para cada objeto String ls na listaStrings
     System.out.println(ls);        // imprime o objeto ls
     if (ls.equals("POO"))          // se o objeto ls for igual a "POO"
       System.out.println(ls);      // imprime novamente o objeto
  }
}
```

Iteração de Coleções – E alterações

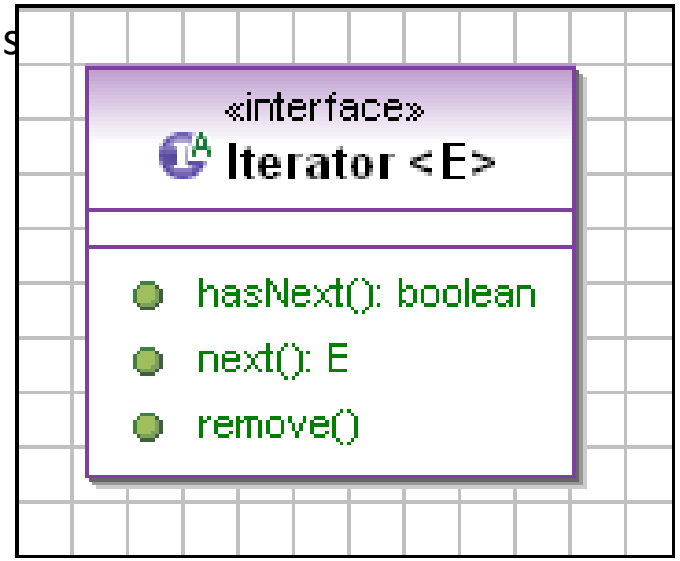
- Note que a introdução de alterações (**`add`**, **`remove`**) numa coleção durante a sua iteração não é segura e normalmente dá origem a uma exceção do tipo:
  - **`ConcurrentModificationException`**.
- Nenhum dos ciclos (**`for`**, **`do-while`**, **`while`**, **`For-Each`**) deve ser usado quando se pretende alterar a lista durante a sua iteração!

```
public static void main(String[] args) {
  List<String> listaStrings = new ArrayList<>();
  listaStrings.add("AO");
  listaStrings.add("IPOO");
  listaStrings.add("POO");
  listaStrings.add("PV");
  listaStrings.add("POO");
  listaStrings.add("POO");
    for (String ls: listaStrings)
      if (ls.equals("POO"))
        listaStrings.remove(ls);
}
```

```
Exception in thread "main" java.util.ConcurrentModificationException
  at java.util.AbstractList$Itr.checkForComodification(AbstractList.java:372)
  at java.util.AbstractList$Itr.next(AbstractList.java:343)
  at slides.Slide10_foreach.main(Slide10_foreach.java:24)
```

Coleções – Interface Iterator

- A Interface **`Iterator<E>`** define os métodos essenciais para iterar (percorrer) uma coleção:
  - **`hasNext()`** – Determinar se a coleção tem ou não um elemento seguinte.
  - **`next()`** – Devolve o elemento seguinte da iteração
  - **`remove()`** – Remove o último elemento iterado.



- **`Iterator<E>`** – Um objeto de uma classe **`<E>`** que implemente a interface **`Iterator`** e que pode ser usado para iterar qualquer coleção de objetos do tipo **`<E>`**.
  - É a única forma segura de alterar coleções durante a iteração
  - Entenda-se por "iterar" percorrer elemento a elemento uma coleção ou parte dela.
  - Exemplo:

```
Iterator<Pessoa> s1 = listaPessoas.iterator();
```

  - Cria um objeto iterador **`s1`** de objetos da classe **`Pessoa`** para a coleção **`listaPessoas`**

Coleções – Interface Iterator

```
public static List<String> preencherLista(List<String> lista){
  lista.add("IPOO");
  lista.add("POO");
  lista.add("PV");
  lista.add("POO");
  lista.add("POO");
  return lista;
}
```

Coleções – Interface Iterator

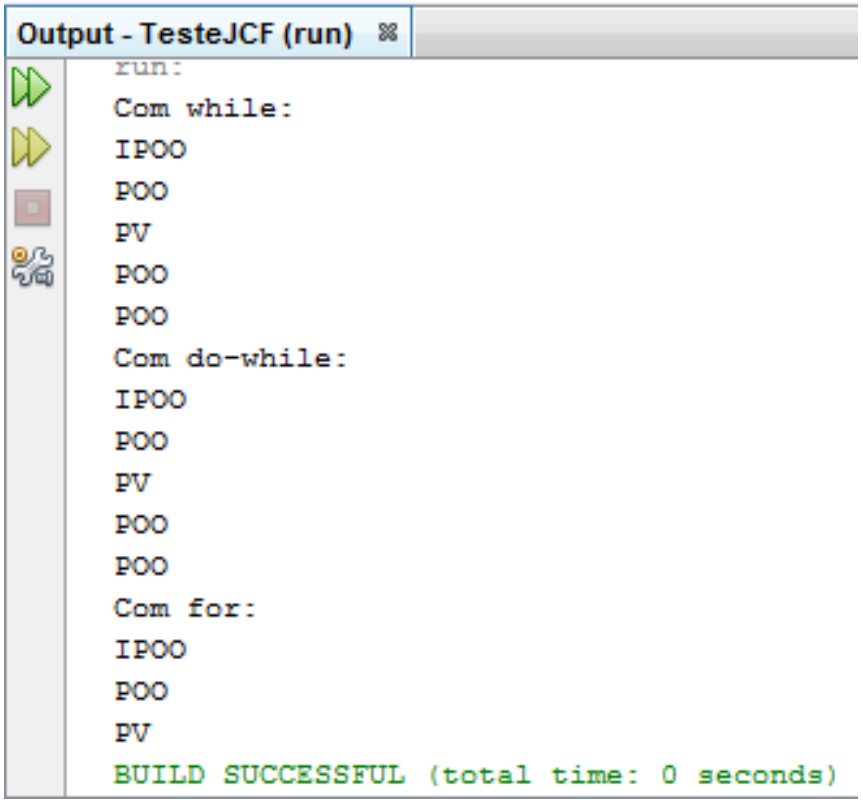
```
public static void main(String[] args) {
   List<String> listaStrings = new ArrayList<>();

   System.out.println("Com while:");
   listaStrings = preencherLista(listaStrings);
   Iterator<String> s1 = listaStrings.iterator();
   while(s1.hasNext()) {
      System.out.println(s1.next());
      s1.remove();
    }

   System.out.println("Com do-while:");
   listaStrings = preencherLista(listaStrings);
   Iterator<String> s2 = listaStrings.iterator();
   if(s2.hasNext()) {
      do {
         System.out.println(s2.next());
         s2.remove();
      } while (s2.hasNext());
   }
```

Coleções – Interface Iterator

```
    System.out.println("Com for:");
    listaStrings = preencherLista(listaStrings);
    Iterator<String> s3 = listaStrings.iterator();
    for (int i=0; i< listaStrings.size(); i++) {
       if (s3.hasNext()) {
          System.out.println(s3.next());
          s3.remove();}
       }
    }
}
```



Equivalência Ciclo For-Each / Iterator

- Um ciclo **`For-Each`**:

```
for (E elemento : colecao) {
   . . .
}
```

- Pode ser implementado utilizando um **`Iterator`**:

```
Iterator<E> iterator = colecao.iterator();
while (iterator.hasNext()) {
   E elemento = iterator.next();
   . . .
}
```

Resumindo

- **Coleção** – agregado de elementos de um mesmo tipo
- JCF – arquitetura que inclui: Interfaces, Classes Abstratas e Concretas, Algoritmos
  - Interfaces genéricas – **`Collection<E>`** - a partir das quais se implementam classes de coleção especificas para guardar objetos de um dado tipo **`<E>`**
  - Os algoritmos da classe **`Collections`** manipulam listas com grande eficiência.
- **Listas**
  - Classe **`ArrayList<E>`** e **`LinkedList<E>`**
- **Iteração e Alterações** em Coleções
  - Ciclo **`For-Each`** (type safe)
  - Não modificar a coleção durante a iteração … a não ser que se use o …
  - **`Iterator<E>`** – a única forma segura de iterar e alterar uma coleção em simultâneo

Leitura Complementar

- Capítulo 8
  - Páginas 253 a 344

- http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html