Computação Orientada a Objetos

Coleções

Profa. Thienne Johnson

Conteúdo

- Java, how to program, 6ª edição
  - Deitel & Deitel
  - Capítulo 19

- Java API – Collections
  - http://java.sun.com/j2se/1.4.2/docs/api/java/util/Collection.html

Coleções

- A estrutura de coleções (**`Collections`** *Framework*) Java fornece componentes reutilizáveis prontos para utilização

- As coleções são padronizadas de modo que aplicativos possam compartilhá-las facilmente, sem preocupação com detalhes de implementação

Coleções (2)

- Com as coleções, os programadores utilizam estruturas de dados existentes, sem se preocupar com a maneira como elas estão implementadas
- É um bom exemplo de reutilização de código

Visão geral

- O que é uma coleção?
  - É uma estrutura de dados (um objeto) que agrupa referências a vários outros objetos
  - algumas vezes chamada de contâiner

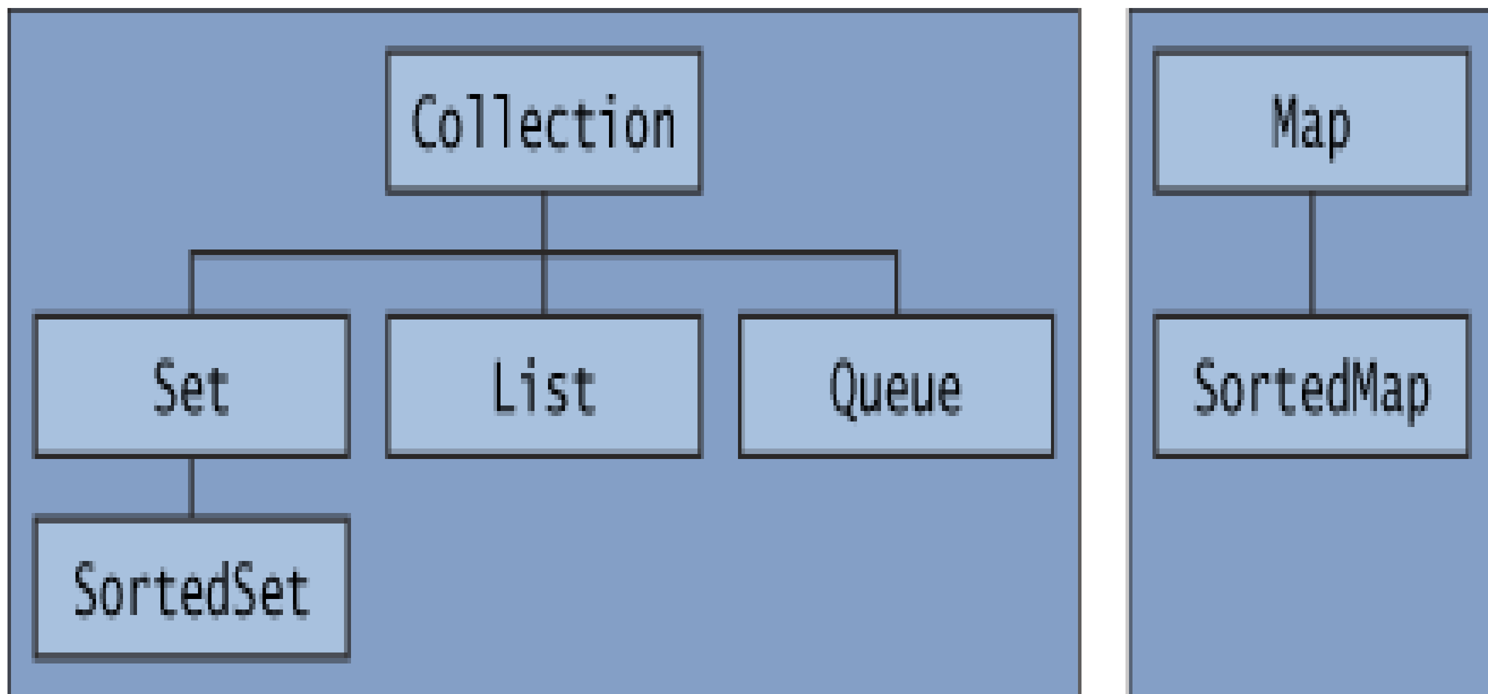
- Usadas para armazenar, recuperar e manipular elementos que formam um grupo natural (objetos do mesmo tipo)

Visão geral

- As interfaces da estrutura de coleções (**`Collections`** *Framework*) Java declaram operações a serem realizadas genericamente em vários tipos de coleções

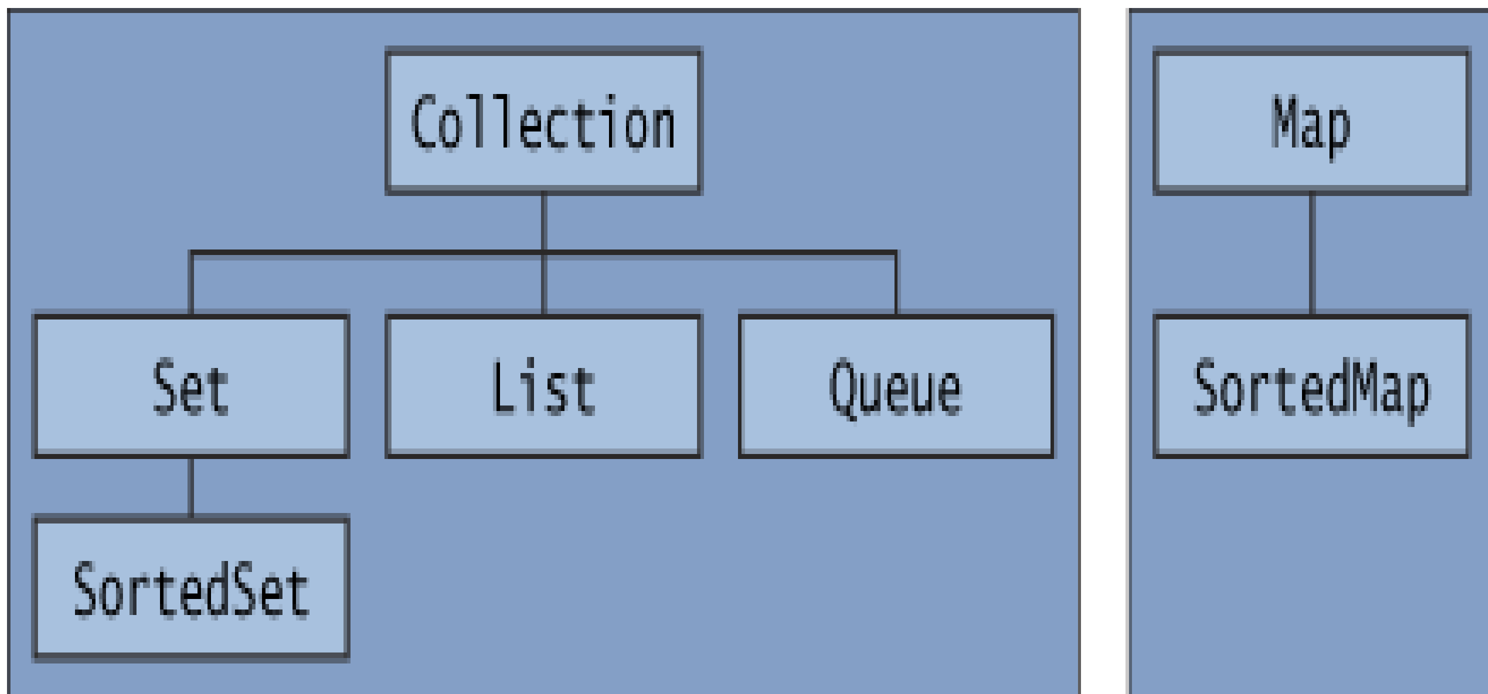
Interfaces da estrutura de coleções

- Interface **Collection**: raiz da hierarquia de coleções



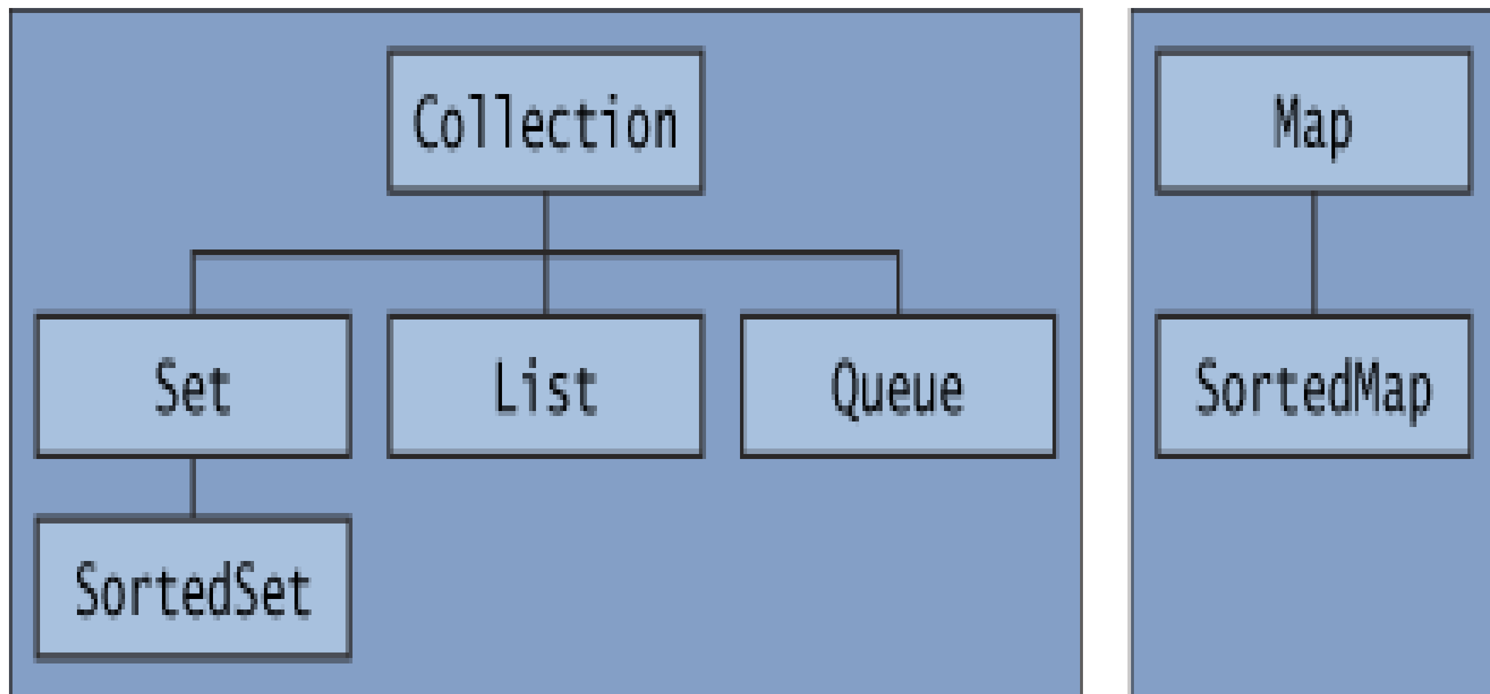
Interfaces da estrutura de coleções (2)

- Interface **Set**: coleção que não contém duplicatas



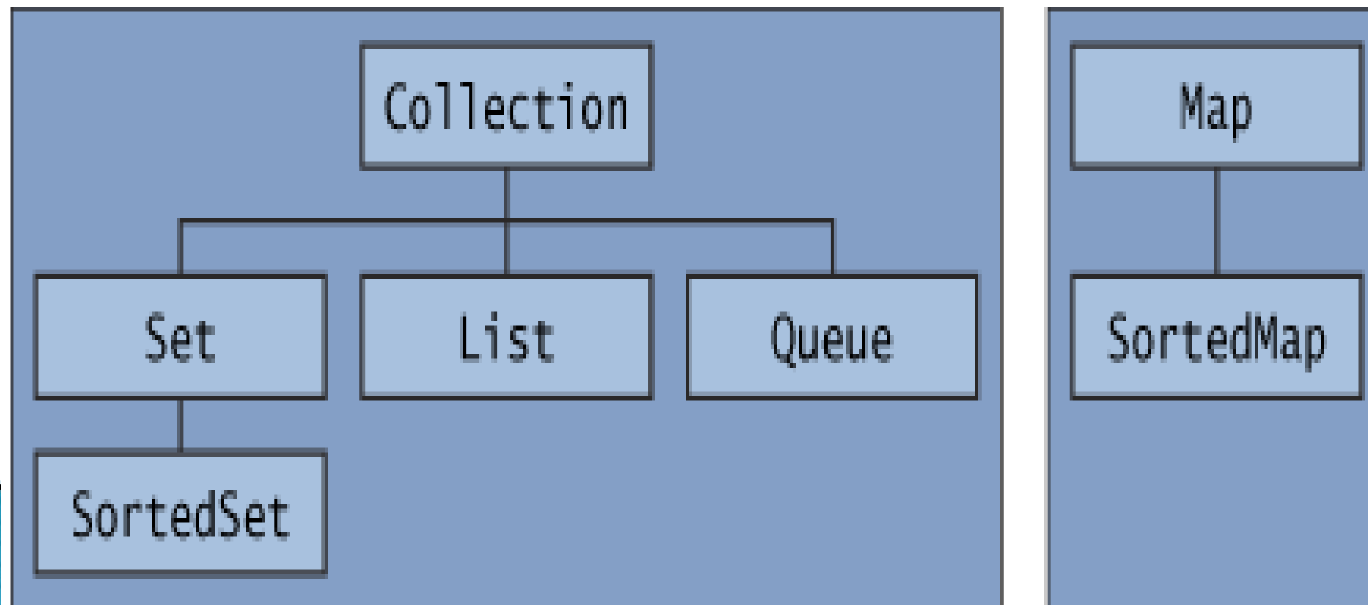
Interfaces da estrutura de coleções (3)

- Interface **List**: coleção que pode conter elementos duplicados



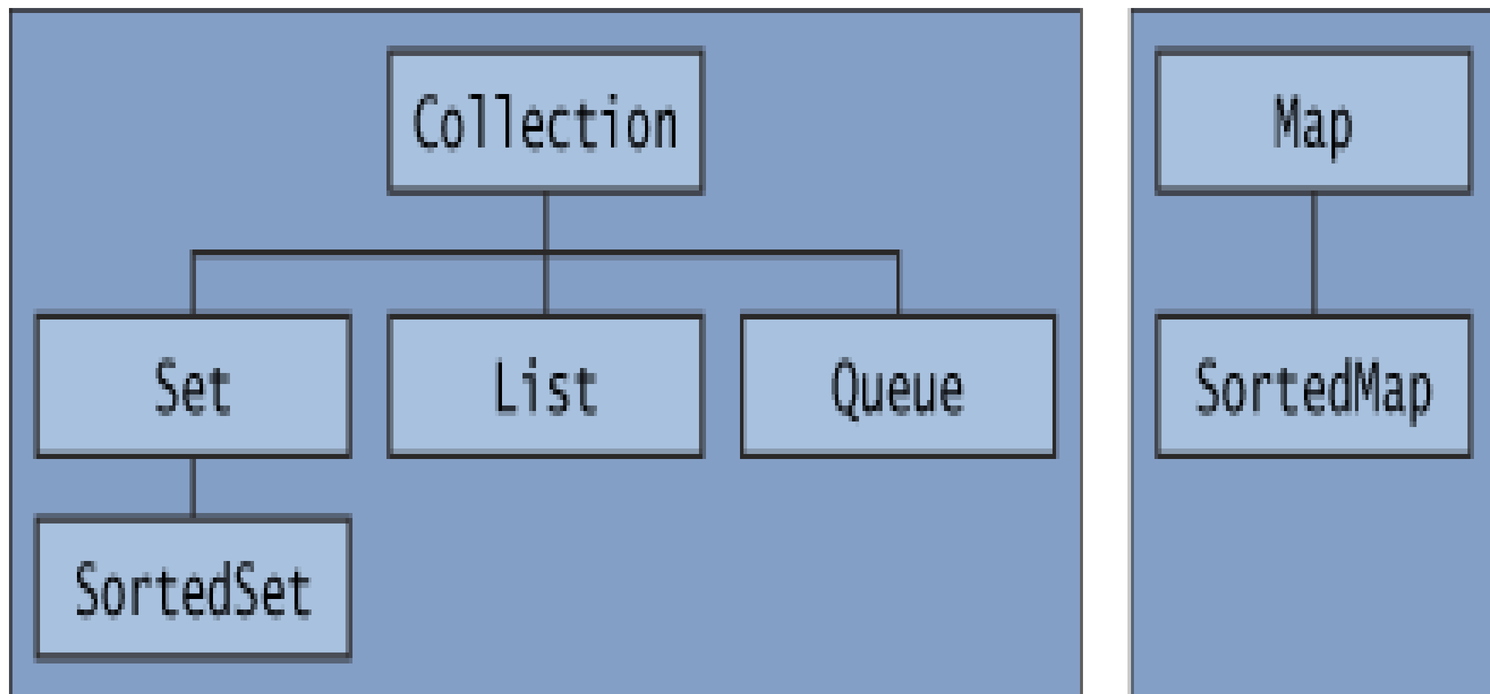
Interfaces da estrutura de coleções (4)

- Interface **Queue**: coleção que modela uma fila de espera (primeiro elemento a entrar, primeiro elemento a sair - *FIFO*)



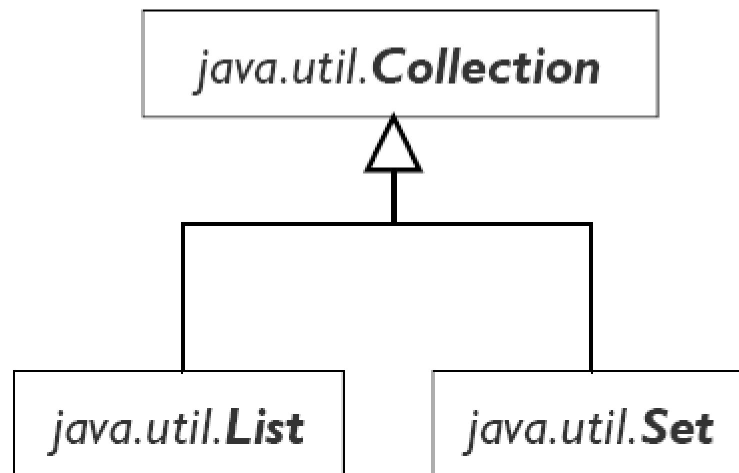
Interfaces da estrutura de coleções (5)

- Interface **Map**: coleção que associa chaves a valores e não pode conter chaves duplicadas



Resumo: principais interfaces

Coleções de
elementos individuais



- seqüência definida
- elementos indexados

- seqüência arbitrária
- elementos não repetem

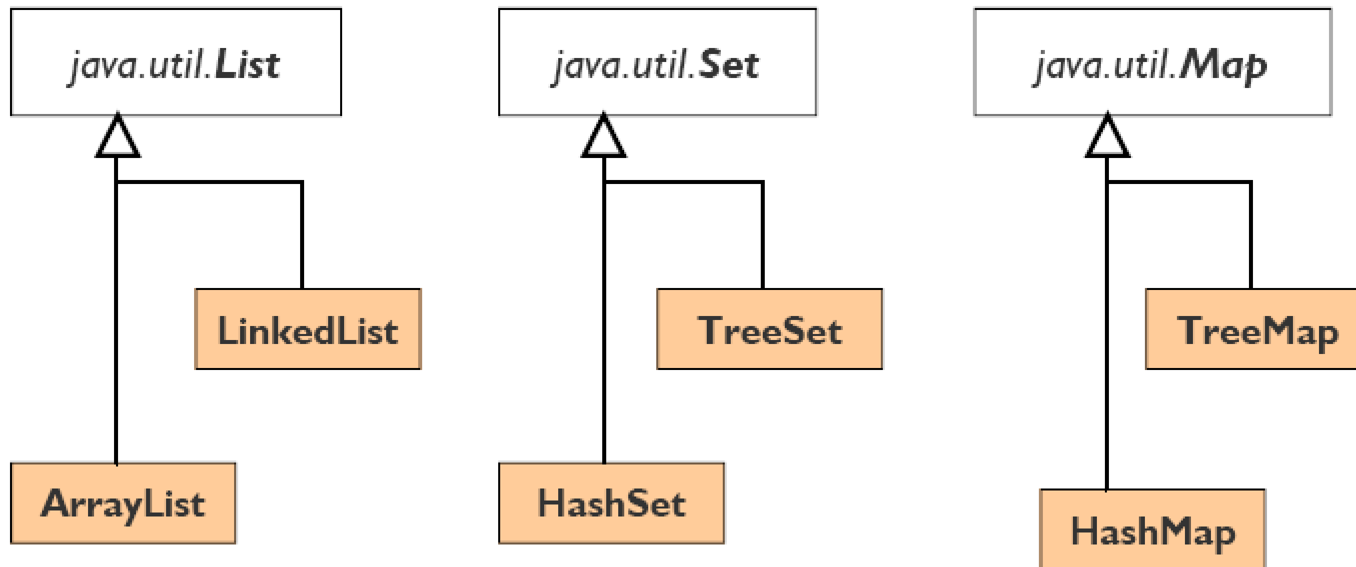
Coleções de
pares de elementos

- Pares chave/valor (vetor associativo)
- **Collection** de valores (podem repetir)
- **Set** de chaves (unívocas)

Interfaces da estrutura de coleções

- Várias implementações para essas interfaces são fornecidas dentro da estrutura de coleções (**`Collections`** *Framework*) Java
- As classes e interfaces da estrutura de coleções são membros do pacote **`java.util`**

Principais implementações



- *Alguns detalhes foram omitidos:*
  - *Classes abstratas intermediárias*
  - *Interfaces intermediárias*
  - *Implementações menos usadas*

Coleções com referências `Object`

- Nas primeiras versões Java, as classes na estrutura de coleções armazenavam e manipulavam referências **`Object`**
- Portanto, era permitido armazenar qualquer objeto em uma coleção

Coleções com referências `Object`

- Um aspecto inconveniente de armazenar referências **`Object`** ocorre ao recuperá-las de uma coleção

- Se um programa precisar processar um tipo específico de objeto, as referências **`Object`** obtidas de uma coleção em geral têm que ser convertidas no tipo apropriado.

Coleções com Object

- Interfaces de coleções baseadas em objetos da classe **Object** permitem que as coleções agrupem qualquer tipo de objeto

```
interface List {
    public void add(Object elemento);
    public Object get(int indice);
    public Iterator iterator();
    ...
}
interface Iterator{
public Object next();
...
}
```

Vantagem: aceita qualquer tipo

Desvantagem: retorna tipo Object que requer coersão, verificada somente em tempo de execução

Coleções com Object

```
public class lista {
    public static void main(String[] args) {
       List list = new ArrayList();
       Integer dado1 = new Integer(10);
       Double dado2 = new Double(10.5);
       for (int i = 1; i <= 50000; i++)
           list.add(0, dado1);
       list.add(0, dado2);
       Iterator it = list.iterator();
       while(it.hasNext())
         Integer dado3 = (Integer) it.next()
    }
    }
```

O compilador não acusa erro, pois Integer e Double são subclasses de Object

Mensagem de erro: Exception in thread "main" java.lang.ClassCastException: java.lang.Double

Coleções com genéricos

- A estrutura de coleções foi aprimorada com as capacidades dos genéricos
- Isso significa que é possível especificar o tipo exato que será armazenado em uma coleção
- Os benefícios da verificação de tipos em tempo de compilação também são obtidos
  - O compilador assegura que os tipos apropriados à coleção estão sendo utilizados

Coleções com genéricos

- Além disso, uma vez que o tipo armazenado em uma coleção é especificado, qualquer referência recuperada dessa coleção terá o tipo especificado

- Isso elimina a necessidade de coerções de tipo explícitas que podem lançar exceções **`ClassCastException`** se o objeto referenciado não for do tipo apropriado.

Interface `Collection`

- Uma coleção representa um grupo de objetos conhecidos como os elementos dessa coleção.

- Interface **`Collection:`** raiz da hierarquia de coleções

- É uma interface genérica
  - Ao declarar uma instância **`Collection`** deve-se especificar o tipo de objeto contido na coleção

Interface `Collection`

- É utilizada para manipular coleções quando deseja-se obter o máximo de generalidade

- Não garante nas implementações
  - Inexistência de duplicatas
  - Ordenação

Interface `Collection` (2)

- Operações básicas: atuam sobre elementos individuais em uma coleção, por ex:
  - adiciona elemento (**`add`**)
  - remove elemento (**`remove`**)
- Operações de volume: atuam sobre todos os elementos de uma coleção, por ex:
  - adiciona coleção (**`addAll`**)
  - remove coleção (**`removeAll`**)
  - mantém coleção (**`retainAll`**)

Interface `Collection` (3)

- A interface Collection também fornece operações para converter uma coleção em um array
  - **`Object[] toArray()`**
  - **`<T> T[] toArray(T[] a)`**

- Além disso, essa interface fornece um método que retorna um objeto **`Iterator`**:
  - permite a um programa percorrer a coleção e remover elementos da coleção durante a iteração

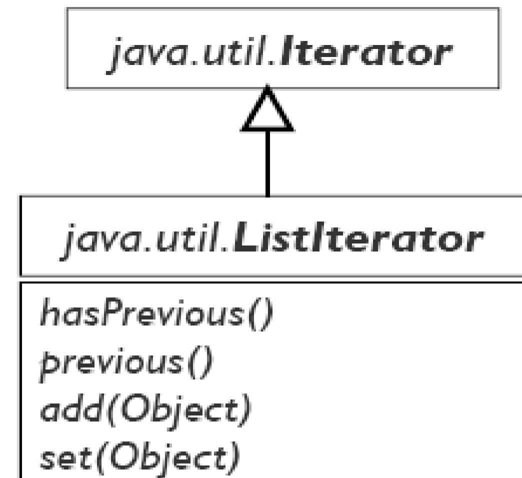
Interface `Collection` (4)

- Outros métodos permitem:
  - determinar quantos elementos pertencem à coleção
    - **`int size()`**
  - determinar se uma coleção está ou não vazia
    - **`boolean isEmpty()`**

Interface Iterator

- *Para navegar dentro de uma Collection e selecionar cada objeto em determinada seqüência*
  - *Uma coleção pode ter vários Iterators*
  - *Isola o tipo da Coleção do resto da aplicação*
  - *Método iterator() (de Collection) retorna Iterator*

```
package java.util;
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```



- *ListIterator possui mais métodos*
  - *Método listIterator() de List retorna ListIterator*

Listas

- Uma coleção **`List`** é uma **`Collection`** que pode conter elementos duplicados

- Como os arrays, os índices de uma coleção **`List`** são baseados em 0 (zero)
  - Isto é, o índice do primeiro elemento é zero

Listas

- Além dos métodos herdados de **`Collection`**, a interface **`List`** fornece métodos para:
  - manipular elementos via seus índices
  - manipular um intervalo específico de elementos
  - procurar elementos
  - obter um **`ListIterator`** para acessar os elementos

Listas

- A interface `List` é implementada por várias classes, incluídas as classes
  - ArrayList
    - Iteração e acesso mais rápidos
  - LinkedList
    - Lista ligada (inclusão & remoção eficiente)
  - Vector
    - Semelhante ao ArrayList (menos eficiente)
    - Uso de métodos *synchronized*

Listas

- A classe **`ArrayList`** e a classe **`Vector`** são implementações de arrays redimensionáveis da interface **`List`**

- A classe **`LinkedList`** é uma implementação de lista encadeada da interface **`List`**

Listas

- O comportamento e as capacidades da classe **ArrayList** são semelhantes às da classe **Vector**

- Entretanto, a classe **Vector** é do Java 1.0, antes de a estrutura de coleções ser adicionada ao Java
  - **Vector** tem vários métodos que não fazem parte da interface **List** e que não estão implementados em **ArrayList**, embora realizem tarefas idênticas

Listas

- Por exemplo, os métodos **`add`** e **`addElement`** da classe **`Vector`** acrescentam um elemento a um objeto **`Vector`**
    - mas somente o método **`add`** é especificado na interface **`List`** e implementado na classe **`ArrayList`**

Listas

- Objetos da classe **`LinkedList`** podem ser utilizados para criar:
  - pilhas
  - filas
  - árvores

List e Iterator

- Tarefa 1: colocar dois arrays de **`String`** em duas listas **`ArrayList`**
  - **`buildList(ArrayList<String>, String[])`**
  - **`printList(ArrayList<String>)`**

- Tarefa 2: utilizar um objeto **`Iterator`** para remover da segunda coleção **`ArrayList`** todos os elementos que também estiverem na primeira coleção
  - **`remove(ArrayList<String>, ArrayList<String>)`**

List e Iterator

- Tarefa 3: adicionar todos elementos de uma coleção em uma segunda coleção
  - **add(List<String>, List<String>)**

- Tarefa 4: Converter cada elemento **String** da lista em letras maiúsculas
  - **upperCase(List<String>)**

- Tarefa 5: Imprime a lista invertida (de trás pra frente)
  - **printReverseList(List<String>)**

ArrayList e Iterator

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class CollectionTest
{
   private static final String[] colors =
      { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
   private static final String[] removeColors =
      { "RED", "WHITE", "BLUE" };
```

ArrayList e Iterator

```
// cria ArrayList, adiciona Colors a ela e a manipula
   public CollectionTest() {
      List< String > list = new ArrayList< String >();
      List< String > removeList = new ArrayList< String >();

      // adiciona elementos no array colors a listar
      for ( String color : colors )
         list.add( color );

      // adiciona elementos de removeColors a removeList
      for ( String color : removeColors )
         removeList.add( color );
```

Arra... ator


```
// cria ArrayList, ...                              ...a
   public CollectionT...() {
      List< String > list = new ArrayList< String >();
      List< String > removeList = new ArrayList< String >();

      // adiciona elementos no array colors a listar
      for ( String color : colors )
         list.add( color );

      // adiciona elementos de removeColors a removeList
      for ( String color : removeColors )
         removeList.add( color );
```

Arra… …ator

Essas duas listas armazenam objetos **String**

```
// cria ArrayList,
   public CollectionTes        {
      List< String > list = new ArrayList< String >();
      List< String > removeList = new ArrayList< String >();

      // adiciona elementos no array colors a listar
      for ( String color : colors )
         list.add( color );

      // adiciona elementos de removeColors a removeList
      for ( String color : removeColors )
         removeList.add( color );
```

ArrayList e Iterator

```
// cria ArrayList, adiciona C
   public CollectionTest() {
      List< String > list = n
      List< String > removeLi

      // adiciona elementos no ar    colors a listar
      for ( String color : colors )
         list.add( color );

      // adiciona elementos de removeColors a removeList
      for ( String color : removeColors )
         removeList.add( color );
```

Preenche a coleção `list` com objetos `String` armazenados no array `color`

ArrayList e Iterator

```
// cria ArrayList, adiciona Colors a ela e a manipula
   public CollectionTest() {
      List< String > list = new ArrayList< String >();
      List< Stri

      // adicio
      for ( Str
         list.a

      // adiciona elementos de removeColors a removeList
      for ( String color : removeColors )
         removeList.add( color );
```

Preenche a coleção **removelist** com objetos **String** armazenados no array **removecolor**

ArrayList e Iterator

```
      System.out.println( "ArrayList: " );

      // gera saída do conteúdo da lista
      for ( int count = 0; count < list.size(); count++ )
         System.out.printf( "%s ", list.get( count ));

      // remove cores contidas em removeList
      removeColors( list, removeList );

      System.out.println("\n\nArrayList after calling
removeColors: " );

      // gera saída do conteúdo da lista
      for ( String color : list )
         System.out.printf( "%s ", color );
   } // fim do construtor CollectionTest
```

Chama o método **`get`** da interface **`List`** para obter cada elemento da lista

```
     Syst

     // gera saída do conte   o da lista
     for ( int count = 0; count < list.size(); count++ )
        System.out.printf( "%s ", list.get( count ));

     // remove cores contidas em removeList
     removeColors( list, removeList );

     System.out.println("\n\nArrayList after calling
removeColors: " );

     // gera saída do conteúdo da lista
     for ( String color : list )
        System.out.printf( "%s ", color );
  } // fim do construtor CollectionTest
```

Array...or

Chama o método **`size`** da interface **`List`** para obter o número de elementos da lista

```
System.out.printl

     // gera saída do conteú      lista
     for ( int count = 0; count < list.size(); count++ )
        System.out.printf( "%s ", list.get( count ));

     // remove cores contidas em removeList
     removeColors( list, removeList );

     System.out.println("\n\nArrayList after calling
 removeColors: " );

     // gera saída do conteúdo da lista
     for ( String color : list )
        System.out.printf( "%s ", color );
  } // fim do construtor CollectionTest
```

ArrayList e Iterator

```
                              ;

                              ista
                              ist.size(); count++ )
              .out.printf( "%s ", list.get( count ));

    // remove cores contidas em removeList
    removeColors( list, removeList );

    System.out.println("\n\nArrayList after calling
removeColors: " );

    // gera saída do conteúdo da lista
    for ( String color : list )
       System.out.printf( "%s ", color );
 } // fim do construtor CollectionTest
```

Uma instrução **for** poderia ter sido utilizada aqui!

ArrayList e Iterator

```
System.out.println( "ArrayList: " );

     // gera saída do conteúdo da lista
                                         st.size(); count++ )
                                         st.get( count ));

     // remove co es contidas em removeList
     removeColors( list, removeList );

     System.out.println("\n\nArrayList after calling
 removeColors: " );

     // gera saída do conteúdo da lista
     for ( String color : list )
        System.out.printf( "%s ", color );
  } // fim do construtor CollectionTest
```

Chamada do método
removeColors

ArrayList e Iterator

Remove de **collection2** as cores (objetos **String**) especificadas em **collection1**

```
private void removeColors( Collection< String >
 collection1, Collection< String > collection2){
     // obtém o iterador
     Iterator< String > iterator = collection1.iterator();

     // loop enquanto a coleção tiver itens
     while (iterator.hasNext())

        if (collection2.contains( iterator.next() ))
           iterator.remove();// remove Color atual
  } // fim do método removeColors
```

ArrayList e Iterator

Permite que quaisquer objetos **`Collections`** que contenham strings sejam passados como argumentos

```
private void removeColors( Collection< String >
 collection1, Collection< String > collection2){
     // obtém o iterador
     Iterator< String > iterator = collection1.iterator();

     // loop enquanto a coleção tiver itens
     while (iterator.hasNext())

        if (collection2.contains( iterator.next() ))
           iterator.remove();// remove Color atual
   } // fim do método removeColors
```

O método acessa os elementos da primeira coleção via um **`Iterator`**. Chama o método **`iterator`** para obter um iterador para **`collection1`**

```
private void re          s( Collection< String >
 collection1, Colle      < String > collection2){
     // obtém o iterado
     Iterator< String > iterator = collection1.iterator();

     // loop enquanto a coleção tiver itens
     while (iterator.hasNext())

        if (collection2.contains( iterator.next() ))
           iterator.remove();// remove Color atual
  } // fim do método removeColors
```

Arra

Observe que os tipos **Collection** e **Iterator** são genéricos!!

```
private void removeColors( Collection< String >
 collection1, Collection< String > collection2){
     // obtém o iterador
     Iterator< String > iterator = collection1.iterator();

     // loop enquanto a coleção tiver itens
     while (iterator.hasNext())

        if (collection2.contains( iterator.next() ))
           iterator.remove();// remove Color atual
  } // fim do método removeColors
```

ArrayList e Iterator

Chama o método **hasnext** da classe **Iterator** para determinar se a coleção tem mais elementos

```
;

   // loop enquanto a coleção tiver itens
   while (iterator.hasNext())

      if (collection2.contains( iterator.next() ))
         iterator.remove();// remove Color atual
} // fim do método removeColors
```

ArrayList e Iterator

O método **hasnext** retorna **true** se outro elemento existir e **false** caso contrário

```
   // loop enquanto a coleção tiver itens
   while (iterator.hasNext())

      if (collection2.contains( iterator.next() ))
         iterator.remove();// remove Color atual
} // fim do método removeColors
```

ArrayList e Iterator

```
private void removeColors( Collection< String >
 collection1, Collection< String > collection2){
```

Chama método **next** da classe **Iterator** para obter uma referência ao próximo elemento da coleção

```
    while (i...e())

        if (collection2.contains( iterator.next() ))
            iterator.remove();// remove Color atual
  } // fim do método removeColors
```

ArrayList e Iterator

```
private void removeColors( Collection< String >
 collection1, Collection< String > collection2){

    while (ite          ext())

       if (collection2.contains( iterator.next() ))
          iterator.remove();// remove Color atual
 } // fim do método removeColors
```

Utiliza o método **contains** da segunda coleção para determinar se a mesma contém o elemento retornado por **next**

Outros métodos de ArrayList

http://java.sun.com/j2se/1.4.2/docs/api/java/util/ArrayList.html

boolean	addAll(int index, Collection c) Inserts all of the elements in the specified Collection into this list, starting at the specified position.
void	clear() Removes all of the elements from this list.
void	ensureCapacity(int minCapacity) Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
boolean	isEmpty() Tests if this list has no elements.
int	lastIndexOf(Object elem) Returns the index of the last occurrence of the specified object in this list.
Object	remove(int index) Removes the element at the specified position in this list.
Object	set(int index, Object element) Replaces the element at the specified position in this list with the specified element.
int	size() Returns the number of elements in this list.
Object[]	toArray() Returns an array containing all of the elements in this list in the correct order.
void	trimToSize() Trims the capacity of this ArrayList instance to be the list's current size.

Erro de programação comum

- Se uma coleção for modificada por um de seus métodos depois de um iterador ter sido criado para essa coleção:
  - o iterador se torna imediatamente inválido!

LinkedList - Exemplo

```
import java.util.List;
import java.util.LinkedList;
import java.util.ListIterator;
public class ListTest {
   private static final String colors[] = { "black", "yellow",
     "green", "blue", "violet", "silver" };
   private static final String colors2[] = { "gold", "white",
     "brown", "blue", "gray", "silver" };

   // configura e manipula objetos LinkedList
   public ListTest()
   {
      List< String > list1 = new LinkedList< String >();
      List< String > list2 = new LinkedList< String >();
```

LinkedList – Exemplo

```
public class ListTest
{
   private static final String colors[] = { "black", "yellow",
      "green", "blue", "violet", "silver" };
   private static final String colors2[] = { "gold", "white",
      "brown", "blue", "gray", "silver" };

   // configura e manipula objetos LinkedList
   public ListTest()
   {
      List< String > list1 = new LinkedList< String >();
      List< String > list2 = new LinkedList< String >();
```

Cria duas listas **`LinkedList`** contendo elementos **`String`**

LinkedList – Exemplo

Adiciona elementos às duas listas

```
 // adiciona elementos a list1
       for ( String color : colors )
          list1.add( color );

       // adiciona elementos a list2
       for ( String color : colors2 )
          list2.add( color );
```

LinkedList – Exemplo

Todos elementos da lista **list1** são adicionados à lista **list2**

```
list1.addAll( list2 ); // concatena as listas
printList( list1 ); // imprime elementos list1
```

LinkedList – Exemplo

Chama o método **addall**
da classe **List**

```
list1.addAll( list2 ); // concatena as listas
printList( list1 ); // imprime elementos list1
```

LinkedList – Exemplo

```
list1.addAll( list2 ); // concatena as listas
printList( list1 ); // imprime elementos list1
```

Chama o método **printlist** para gerar a saída do conteúdo de **list1**

LinkedList – Exemplo

Gera saída do conteúdo de **`List`**

```
public void printList(List< String > list)
    {
        System.out.println( "\nlist: " );

        for ( String color : list )
            System.out.printf( "%s ", color );

        System.out.println();
    } // fim do método printList
```

LinkedList – Exemplo

Converte cada elemento **string** da lista em letras maiúsculas

```
convertToUppercaseStrings( list1 );
printList( list1 ); // imprime elementos list1
```

Chama o método **printlist** para gerar a saída do conteúdo de **list1**

LinkedList – Exemplo

Localiza objetos **String** e
converte em letras maiúsculas

```
private void convertToUppercaseStrings(List< String > list){
      ListIterator< String > iterator = list.listIterator();

      while (iterator.hasNext())
      {
         String color = iterator.next();  // obtém o item
         iterator.set(color.toUpperCase() ); // converte em
    letras maiúsculas
      } // fim do while
   } // fim do método convertToUppercaseStrings
```

LinkedList – Exemplo

Chama o método **`listIterator`** da interface **`List`** para obter um iterador para a lista

```
private void convertToUppercaseStrings(List< String > list){
      ListIterator< String > iterator = list.listIterator();

      while (iterator.hasNext())
      {
         String color = iterator.next();  // obtém o item
         iterator.set( color.toUpperCase() ); // converte em
    letras maiúsculas
      } // fim do while
   } // fim do método convertToUppercaseStrings
```

LinkedList – Exemplo

```
private void convertToUppercaseStrings(List< String >
     list){
       ListIterator< String > iterator =
     list.listIterator();

       while (iterator.hasNext())
       {
          String color = iterator.next();  // obtém o item
          iterator.set( color.toUpperCase() ); // converte
     em letras maiúsculas
       } // fim do whi
    } // fim do                  ToUppercaseStrings
```

Chama o método **`toUppercase`** da classe **`String`**
para obter uma versão em letras maiúsculas da **`string`**

LinkedList – Exemplo

```
private void convertToUppercaseStrings(List< String >
     list){
```

Chama o método **set** da classe **ListIterator** para substituir a **string** referenciada pelo iterador pela sua versão em letras maiúsculas

```
       {
          String co  r = iterator.next();  // obtém o item
          iterator.set( color.toUpperCase() ); // converte
     em letras maiúsculas
       } // fim do while
     } // fim do método convertToUppercaseStrings
```

LinkedList – Exemplo

Chama o método **`removeItems`** para remover os elementos que iniciam no índice 4 até, mas não incluindo o índice 7 da lista

```
System.out.print( "\nremovendo os elementos de 4 a 6..." );
removeItems( list1, 4, 7 );
printList( list1 ); // imprime elementos list1
```

Chama o método **`printlist`** para gerar a saída do conteúdo de **`list1`**

LinkedList – Exemplo

```
private void removeItems(List< String > list, int start, int
      end)
    {
       list.subList( start, end ).clear();  // remove os itens
    } // fim do método removeItems
```

Chama o método **`sublist`** da classe **`List`** para obter uma parte da lista

LinkedList – Exemplo

```
private void removeItems(List< String > list, int start, int end)
   {
      list.subList( start, end ).clear();  // remove os itens
   } // fi      étodo removeItems
```

O método **sublist** aceita dois argumentos:
os índices inicial e final da sublista
Obs: o índice final não faz parte da sublista!

LinkedList – Exemplo

```
private void removeItems(List< String > list, int start, int
     end)
   {
      list.subList( start, end ).clear();  // remove os itens
   } // fim do método remov
```

Chama o método **clear** da classe **List** para remover todos os elementos da sublista contida na lista **list**

LinkedList – Exemplo

Imprime a lista na ordem inversa

```
printReversedList( list1 );
```

LinkedList – Exemplo

Imprime a lista invertida
(de trás pra frente)

```
private void printReversedList(List< String > list){
    ListIterator< String > iterator = list.listIterator( list.size());

    System.out.println( "\nReversed List:" );

    // imprime lista na ordem inversa
     while (iterator.hasPrevious())
        System.out.printf( "%s ", iterator.previous());
  } // fim do método printReversedList
```

LinkedList – Exemplo

Chama o método **`listIterator`** da classe **`List`** com um argumento que especifica a posição inicial do iterador (nesse caso, o último elemento)

```
private void printReversedList(List< String > list){
    ListIterator< String > iterator = list.listIterator(list.size());

    System.out.println( "\nReversed List:" );

    // imprime lista na ordem inversa
     while (iterator.hasPrevious())
        System.out.printf( "%s ", iterator.previous());
  } // fim do método printReversedList
```

LinkedList – Exemplo

```
private void printReversedList(List< String > list){
    Li
    li

    Sy

    // imprime lista na ordem inversa
     while (iterator.hasPrevious())
        System.out.printf( "%s ", iterator.previous());
  } // fim do método printReversedList
```

Chama o método **hasprevious** da classe **ListIterator** para determinar se há mais elementos ao percorrer a lista em ordem invertida

LinkedList – Exemplo

```
private void printReversedList(List< String > list){
    ListIterator< String > iterator = list.listIterator(
    list.size() );

    System.out.println( "\nReversed List:" );

    // imprime lista na ordem inversa
     while (iterator.hasPrevious())
        System.out.printf( "%s ", iterator.previous());
  } // fim do método printReversedList
```

Chama o método **`previous`** da classe **`ListIterator`** para obter o elemento anterior da lista

Outros Métodos de List

http://java.sun.com/j2se/1.4.2/do

void **clear**()
Removes all of the elements from this list (optional operation).

boolean **contains**(Object o)
Returns true if this list contains the specified element.

Object **get**(int index)
Returns the element at the specified position in this list.

int **indexOf**(Object o)
Returns the index in this list of the first occurrence of the specified element, or -1 if this list does not contain this element.

boolean **isEmpty**()
Returns true if this list contains no elements.

Object **remove**(int index)
Removes the element at the specified position in this list (optional operation).

boolean **removeAll**(Collection c)
Removes from this list all the elements that are contained in the specified collection

Object **set**(int index, Object element)
Replaces the element at the specified position in this list with the specified element

int **size**()
Returns the number of elements in this list.

List **subList**(int fromIndex, int toIndex)
Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.

Object[] **toArray**()
Returns an array containing all of the elements in this list in proper sequence.

Códigos fonte dos exemplos

- http://wps.prenhall.com/wps/media/objects/2238/2292414/ch19.zip