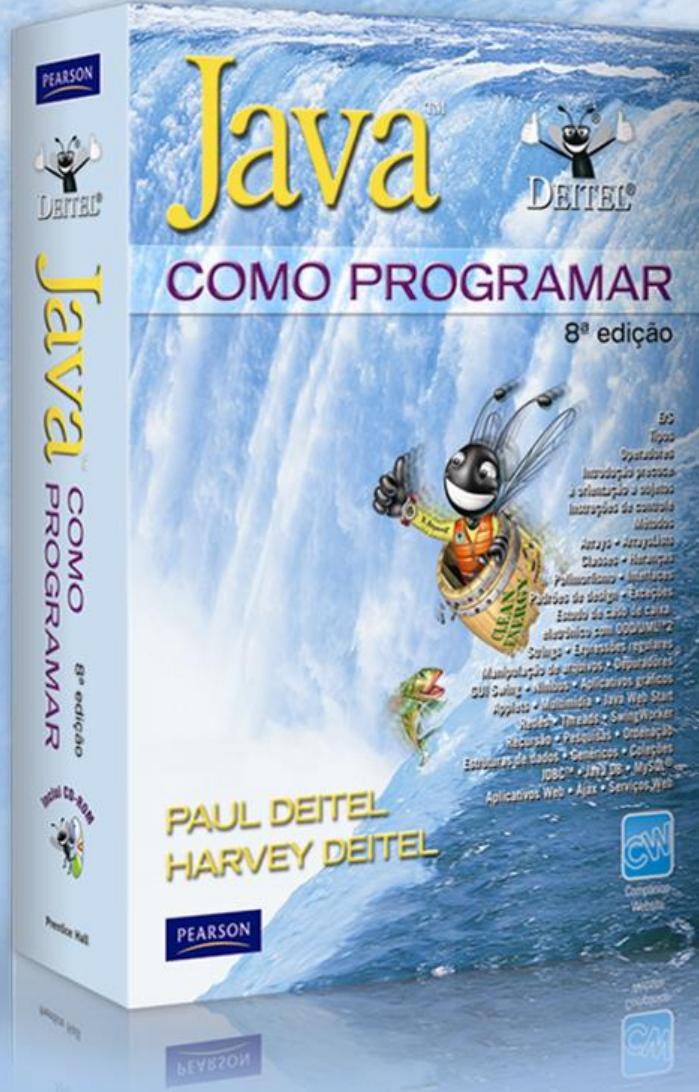


Capítulo 20

Coleções genéricas

Java Como Programar, 8/E





COMO PROGRAMAR

8^a edição

OBJETIVOS

Neste capítulo, você aprenderá:

- O que são coleções.
- A utilizar a classe `Arrays` para manipulações de array.
- A formar estruturas de dados encadeadas utilizando referências, classes autorreferenciais e recursão.
- A reconhecer as classes empacotadoras de tipo que permitem aos programas processar valores de dados primitivos como objetos.
- A utilizar implementações de estrutura de coleções (estrutura de dados pré-construída).
- A utilizar métodos de estrutura de coleções (como `search`, `sort` e `fill`) para manipular coleções.
- A utilizar as interfaces de estrutura de coleções para programar com coleções polimorficamente.
- A utilizar iteradores para “percorrer” uma coleção.
- A utilizar tabelas de hash persistentes manipuladas com objetos da classe `Properties`.
- A utilizar empacotadores de sincronização e de modificabilidade.



COMO PROGRAMAR

8^a edição

20.1 Introdução

20.2 Visão geral das coleções

20.3 Classes empacotadoras de tipo para tipos primitivos

20.4 Autoboxing e auto-unboxing

20.5 Interface Collection e classe Collections

20.6 Listas

20.6.1 ArrayList e Iterator

20.6.2 LinkedList

20.7 Métodos de coleções

20.7.1 Método sort

20.7.2 Método shuffle

20.7.3 Métodos reverse, fill, copy, max e min

20.7.4 Método binarySearch

20.7.5 Métodos addAll, frequency e disjoint



COMO PROGRAMAR

8^a edição

- 20.8** Classe Stack do pacote `java.util`
- 20.9** Classe PriorityQueue e Interface Queue
- 20.10** Conjuntos
- 20.11** Mapas
- 20.12** Classe Properties
- 20.13** Coleções sincronizadas
- 20.14** Coleções não modificáveis
- 20.15** Implementações abstratas
- 20.16** Conclusão



COMO PROGRAMAR

8^a edição

20.1 Introdução

- ▶ **Estrutura de coleções** do Java.
- ▶ Estruturas de dados predefinidas.
- ▶ Interfaces e métodos para manipular essas estruturas de dados



COMO PROGRAMAR

8^a edição

20.2 Visão geral das coleções

- ▶ Uma **coleção** é uma estrutura de dados — na realidade, um objeto — que pode armazenar referências a outros objetos.
- ▶ Normalmente, as coleções contêm referências a objetos que são todos do mesmo tipo.
- ▶ A Figura 20.1 lista algumas interfaces da estrutura de coleções.
- ▶ Pacote `java.util`.



COMO PROGRAMAR

8^a edição

Interface	Descrição
Collection	A interface-raiz na hierarquia de coleções a partir da qual as interfaces Set, Queue e List são derivadas.
Set	Uma coleção que não contém duplicatas.
List	Uma coleção ordenada que pode conter elementos duplicados.
Map	Associa chaves a valores e não pode conter chaves duplicadas.
Queue	Em geral, uma coleção primeiro a entrar, primeiro a sair que modela uma fila de espera; outras ordens podem ser especificadas.

Figura 20.I | Algumas interfaces da estrutura de coleções.



COMO PROGRAMAR

8^a edição

20.3 Classes empacotadoras de tipo para tipos primitivos

- ▶ Cada tipo primitivo tem uma **classe empacotadora de tipo** correspondente (no pacote `java.lang`).

Boolean, Byte, Character, Double, Float, Integer, Long e Short.

- ▶ Cada classe empacotadora de tipo permite manipular valores de tipo primitivo como objetos.
- ▶ As coleções não podem manipular variáveis de tipos primitivos.
- ▶ Elas podem manipular objetos das classes empacotadoras de tipos, porque toda classe, em última instância, deriva de `Object`.



COMO PROGRAMAR

8^a edição

- ▶ Cada uma das classes empacotadoras de tipos numéricos — **Byte**, **Short**, **Integer**, **Long**, **Float** e **Double** — estendem a classe **Number**.
- ▶ As classes empacotadoras de tipos são classes **final**, portanto, você não pode estendê-las.
- ▶ Os tipos primitivos não têm métodos, então, os métodos relacionados a um tipo primitivo estão localizados na classe empacotadora de tipo correspondente.



COMO PROGRAMAR

8^a edição

20.4 Autoboxing e auto-unboxing

- ▶ Uma **conversão boxing** converte um valor de um tipo primitivo em um objeto da classe empacotadora de tipo correspondente.
- ▶ Uma **conversão unboxing** converte um objeto de uma classe empacotadora de tipo em um valor do tipo primitivo correspondente.
- ▶ Essas conversões podem ser realizadas automaticamente (chamam-se **autoboxing** e **auto-unboxing**).
- ▶ Exemplo:

```
// cria integerArray
Integer[] integerArray = new Integer[ 5 ];

// assign Integer 10 to integerArray[ 0 ]
integerArray[ 0 ] = 10;

// get int value of Integer int value = integerArray[
0 ];
```



COMO PROGRAMAR

8^a edição

20.5 Interface Collection e classe Collections

- ▶ A interface **Collection** é a interface-raiz a partir da qual as interfaces **Set**, **Queue** e **List** são derivadas.
- ▶ A interface **Set** define uma coleção que não contém duplicatas.
- ▶ A interface **Queue** define uma coleção que representa uma fila de espera.
- ▶ A interface **Collection** contém **operações de volume** para adicionar, limpar e comparar objetos de uma coleção.
- ▶ Uma **Collection** também pode ser convertida em um array.
- ▶ A interface **Collection** fornece um método que retorna um objeto **Iterator**, que permite a um programa percorrer a coleção e remover elementos da coleção durante a iteração.
- ▶ A classe **Collections** fornece os métodos **static** que pesquisam, classificam e realizam outras operações em coleções.



COMO PROGRAMAR

8^a edição



Observação de engenharia de software 20.I

Collection é comumente utilizada como um tipo de parâmetro nos métodos para permitir processamento polimórfico de todos os objetos que implementam a interface Collection.



COMO PROGRAMAR

8^a edição



Observação de engenharia de software 20.2

A maioria das implementações de coleção fornece um construtor que aceita um argumento Collection, permitindo, assim, que uma nova coleção a ser construída contenha os elementos da coleção especificada.



COMO PROGRAMAR

8^a edição

20.6 Listas

- ▶ Uma **List** (às vezes, chamada **sequência**) é uma **Collection** que pode conter elementos duplicados.
- ▶ Os índices de **List** são baseados em zero.
- ▶ Além dos métodos herdados de **Collection**, **List** fornece métodos para manipular elementos via seus índices, manipular um intervalo especificado de elementos, procurar elementos e obter um **ListIterator** para acessar os elementos.
- ▶ A interface **List** é implementada por várias classes, incluindo **ArrayList**, **LinkedList** e **Vector**.
- ▶ O autoboxing ocorre quando você adiciona valores de tipo primitivo a objetos dessas classes, pois eles armazenam apenas referências a objetos.



COMO PROGRAMAR

8^a edição

- ▶ As classes `ArrayList` e `Vector` são implementações de arrays redimensionáveis de `List`.
- ▶ Inserir um elemento entre elementos existentes de um `ArrayList` ou `Vector` é uma operação ineficiente.
- ▶ Uma `LinkedList` permite inserção (ou remoção) eficiente de elementos no meio de uma coleção.
- ▶ Discutimos a arquitetura de listas encadeadas (vinculadas) no Capítulo 22.
- ▶ A principal diferença entre `ArrayList` e `Vector` é que `Vectors` são sincronizados por padrão, enquanto `ArrayLists` não o são.
- ▶ As coleções não sincronizadas fornecem melhor desempenho que as sincronizadas.
- ▶ Por essa razão, `ArrayList` é, em geral, preferida a `Vector` em programas que não compartilham uma coleção entre threads.



COMO PROGRAMAR

8^a edição



Dica de desempenho 20.I

ArrayLists comportam-se como Vectors sem sincronização e, portanto, executam mais rápido que Vectors porque ArrayLists não tem o overhead de sincronização de thread.



COMO PROGRAMAR

8^a edição



Observação de engenharia de software 20.3

LinkedLists podem ser utilizadas para criar pilhas, filas e dequees (double-ended queues — filas com dupla terminação). A estrutura de coleções fornece implementações de algumas dessas estruturas de dados.



COMO PROGRAMAR

8^a edição

20.6.1 ArrayList e Iterator

- ▶ O método **List add** adiciona um item ao final de uma lista.
- ▶ O método **List size** retorna o número de elementos.
- ▶ O método **List get** recupera o valor de um elemento individual de um índice especificado.
- ▶ O método **Collection iterator** obtém um **Iterator** de uma **Collection**.
- ▶ O método **Iterator hasNext** determina se uma **Collection** contém mais elementos. Retorna **true** se houver outro elemento e, **false**, caso contrário.
- ▶ O método **Iterator next** retorna uma referência para o próximo elemento.
- ▶ O método **Collection contains** determina se uma **Collection** contém um elemento especificado.
- ▶ O método **Iterator remove** remove o elemento atual de uma **Collection**.



COMO PROGRAMAR

8^a edição

```
1 // Figura 20.2: CollectionTest.java
2 // A interface Collection demonstrada via um objeto ArrayList.
3 import java.util.List;
4 import java.util.ArrayList;
5 import java.util.Collection;
6 import java.util.Iterator;
7
8 public class CollectionTest
9 {
10     public static void main( String[] args )
11     {
12         // adiciona elementos no array colors a listar
13         String[] colors = { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
14         List< String > list = new ArrayList< String >(); ←
15
16         for ( String color : colors )
17             list.add( color ); // adiciona a cor ao fim da lista
18
19         // adiciona elementos no array removeColors em removeList
20         String[] removeColors = { "RED", "WHITE", "BLUE" };
21         List< String > removeList = new ArrayList< String >();
22 }
```

É uma boa prática referenciar uma coleção via uma variável de tipo de interface - é mais fácil alterar a coleção posteriormente

Figura 20.2 | A interface Collection demonstrada via um objeto ArrayList. (Parte 1 de 3.)



COMO PROGRAMAR

8^a edição

```
23     for ( String color : removeColors )
24         removeList.add( color );
25
26     // gera a saída do conteúdo da lista
27     System.out.println( "ArrayList: " );
28
29     for ( int count = 0; count < list.size(); count++ )
30         System.out.printf( "%s ", list.get( count ) );
31
32     // remove da lista as cores contidas em removeList
33     removeColors( list, removeList );
34
35     // gera a saída do conteúdo da lista
36     System.out.println( "\n\nArrayList after calling removeColors: " );
37
38     for ( String color : list )
39         System.out.printf( "%s ", color );
40 } // fim de main
41
```

Figura 20.2 | A interface Collection demonstrada via um objeto ArrayList. (Parte 2 de 3.)



COMO PROGRAMAR

8^a edição

```
42 // remove cores especificadas em collection2 a partir de collection1
43 private static void removeColors(Collection< String > collection1,
44     Collection< String > collection2 )
45 {
46     // obtém o iterador
47     Iterator< String > iterator = collection1.iterator();
48
49     // loop enquanto a coleção tiver itens
50     while (iterator.hasNext() )
51     {
52         if ( collection2.contains( iterator.next() ) )
53             iterator.remove(); // remove Color atual
54     } // fim do while
55 } // fim do método removeColors
56 } // fim da classe CollectionTest
```

O método funciona com
qualquer Collection

ArrayList:

MAGENTA RED WHITE BLUE CYAN

ArrayList after calling removeColors:

MAGENTA CYAN

Figura 20.2 | A interface Collection demonstrada via um objeto ArrayList. (Parte 3 de 3.)



COMO PROGRAMAR

8^a edição



Erro comum de programação 20.1

Se uma coleção for modificada por um dos seus métodos depois que um iterador é criado para essa coleção, o iterador torna-se imediatamente inválido — as operações realizadas com o iterador depois desse ponto lançam ConcurrentModificationExceptions. Por essa razão, diz-se que os iteradores “respondem rápido”.



COMO PROGRAMAR

8^a edição

20.6.2 LinkedList

- ▶ O método **List addAll** acrescenta todos os elementos de uma coleção ao final de uma **List**.
- ▶ O método **List listIterator** obtém o **iterador bidirecional** de uma **List**.
- ▶ O método **String toUpperCase** obtém uma versão em letras maiúsculas de uma **String**.
- ▶ O método **ListIterator set** substitui o elemento atual ao qual o iterador referencia por um objeto especificado.
- ▶ O método **String toLowerCase** retorna uma versão em letras minúsculas de uma **String**.
- ▶ O método **List subList** obtém uma parte de uma **List**.
- ▶ Isso é o que chamamos de **método de visualização de intervalo**, que permite ao programa visualizar uma parte da lista.



COMO PROGRAMAR

8^a edição

- ▶ O método **List clear** remove os elementos de uma **List**.
- ▶ O método **List size** retorna o número de itens na **List**.
- ▶ O método **ListIterator hasPrevious** determina se há mais elementos ao percorrer a lista de trás para frente.
- ▶ O método **ListIterator previous** obtém o elemento anterior na lista.



COMO PROGRAMAR

8^a edição

```
1 // Figura 20.3: ListTest.java
2 // Lists, LinkedLists e ListIterators.
3 import java.util.List;
4 import java.util.LinkedList;
5 import java.util.ListIterator;
6
7 public class ListTest
8 {
9     public static void main( String[] args )
10    {
11        // adiciona elementos colors a list1
12        String[] colors =
13            { "black", "yellow", "green", "blue", "violet", "silver" };
14        List< String > list1 = new LinkedList< String >();
15
16        for ( String color : colors )
17            list1.add( color );
18
19        // adiciona elementos colors2 à list2
20        String[] colors2 =
21            { "gold", "white", "brown", "blue", "gray", "silver" };
22        List< String > list2 = new LinkedList< String >();
23    }
```

Figura 20.3 | Lists, LinkedLists e ListIterators. (Parte I de 4.)



COMO PROGRAMAR

8^a edição

```
24     for ( String color : colors2 )
25         list2.add( color );
26
27     list1.addAll( list2 ); // concatena as Listas
28     list2 = null; // libera recursos
29     printList( list1 ); // imprime elementos list1
30
31     convertToUppercaseStrings( list1 ); // converte para string maiúscula
32     printList( list1 ); // imprime elementos list1
33
34     System.out.print( "\nDeleting elements 4 to 6..." );
35     removeItems( list1, 4, 7 ); // remove itens 4-6 da lista
36     printList( list1 ); // imprime elementos list1
37     printReversedList( list1 ); // imprime lista na ordem inversa
38 } // fim de main
39
40 // gera saída do conteúdo de List
41 private static void printList(List< String > list )
42 {
43     System.out.println( "\nlist: " );
44
45     for ( String color : list )
46         System.out.printf( "%s ", color );
47 }
```

Figura 20.3 | Lists, LinkedLists e ListIterators. (Parte 2 de 4.)



COMO PROGRAMAR

8^a edição

```
48     System.out.println();
49 } // fim do método printList
50
51 // localiza objetos String e converte em letras maiúsculas
52 private static void convertToUppercaseStrings(List< String > list )
53 {
54     ListIterator< String > iterator = list.listIterator();
55
56     while (iterator.hasNext() )
57     {
58         String color = iterator.next(); // obtém o item
59         iterator.set( color.toUpperCase() ); // converte em letras maiúsculas
60     } // fim do while
61 } // fim do método convertToUppercaseStrings
62
63 // obtém sublistas e utiliza método clear para excluir itens da sublistas
64 private static void removeItems(List< String > list,
65     int start, int end )
66 {
67     list.subList( start, end ).clear(); // remove os itens
68 } // fim do método removeItems
69
```

subList retorna uma
visualização em uma List

Figura 20.3 | Lists, LinkedLists e ListIterators. (Parte 3 de 4.)



COMO PROGRAMAR

8^a edição

```
70     // imprime lista invertida
71     private static void printReversedList(List< String > list )
72     {
73         ListIterator< String > iterator = list.listIterator( list.size() );
74
75         System.out.println( "\nReversed List:" );
76
77         // imprime lista na ordem inversa
78         while (iterator.hasPrevious() )
79             System.out.printf( "%s ", iterator.previous() );
80     } // fim do método printReversedList
81 } // fim da classe ListTest
```

```
list:
black yellow green blue violet silver gold white brown blue gray silver

list:
BLACK YELLOW GREEN BLUE VIOLET SILVER GOLD WHITE BROWN BLUE GRAY SILVER

Deleting elements 4 to 6...
list:
BLACK YELLOW GREEN BLUE WHITE BROWN BLUE GRAY SILVER

Reversed List:
SILVER GRAY BLUE BROWN WHITE BLUE GREEN YELLOW BLACK
```

Figura 20.3 | Lists, LinkedLists e ListIterators. (4 de 4.)



COMO PROGRAMAR

8^a edição

- ▶ A classe **Arrays** fornece o método **static asList** para visualizar um array como uma coleção **List**.
- ▶ Uma visualização **List** permite manipular o array como se ele fosse uma lista.
- ▶ Isso é útil para adicionar os elementos de um array a uma coleção e para classificar os elementos do array.
- ▶ Qualquer modificação feita por meio da visualização **List** altera o array, e qualquer modificação feita no array altera a visualização **List**.
- ▶ A única operação permitida na visualização retornada por **asList** é **set**, que altera o valor da visualização e o array de apoio.
- ▶ Quaisquer outras tentativas de alterar a visualização resultam em uma **UnsupportedOperationException**.
- ▶ O método **List toArray** obtém um array de uma coleção **List**.

COMO PROGRAMAR

8^a edição

```
1 // Figura 20.4: UsingToArray.java
2 // Visualizando arrays como Lists e convertendo Lists em arrays.
3 import java.util.LinkedList;
4 import java.util.Arrays;
5
6 public class UsingToArray
7 {
8     // cria uma LinkedList, adiciona elementos e converte em array
9     public static void main( String[] args )
10    {
11        String[] colors = { "black", "blue", "yellow" };
12
13        LinkedList< String > links =
14            new LinkedList< String >( Arrays.asList( colors ) );
15
16        links.addLast( "red" ); // adiciona como o último item
17        links.add( "pink" ); // adiciona ao final
18        links.add( 3, "green" ); // adiciona no terceiro índice
19        links.addFirst( "cyan" ); // adiciona como primeiro item
20
21        // obtém elementos LinkedList como um array
22        colors = links.toArray( new String[ links.size() ] );
23
24        System.out.println( "colors: " );
```

Prealocar o array
permite que toArray
simplesmente copie os
elementos para o array

Figura 20.4 | Visualizando arrays como Lists e convertendo Lists em arrays. (Parte 1 de 2.)



COMO PROGRAMAR

8^a edição

```
25
26     for ( String color : colors )
27         System.out.println( color );
28     } // fim de main
29 } // fim da classe UsingToArray
```

```
colors:
cyan
black
blue
yellow
green
red
pink
```

Figura 20.4 | Visualizando arrays como Lists e convertendo Lists em arrays. (Parte 2 de 2.)



COMO PROGRAMAR

8^a edição

- ▶ O método **LinkedList addLast** adiciona um elemento ao final de uma **List**.
- ▶ O método **LinkedList add** também adiciona um elemento ao final de uma **List**.
- ▶ O método **LinkedList addFirst** adiciona um elemento ao início de uma **List**.



COMO PROGRAMAR

8^a edição



Erro comum de programação 20.2

Passar um array que contém dados para toArray pode causar erros de lógica. Se o número de elementos no array for menor que o número de elementos na lista em que toArray é chamado, um novo array é alocado para armazenar os elementos da lista — sem preservar os elementos do argumento de array. Se o número de elementos no array for maior que o número de elementos na lista, os elementos do array (iniciando no índice zero) serão sobreescritos pelos elementos da lista. Os elementos do array que não são sobreescritos retêm seus valores.



COMO PROGRAMAR

8^a edição

20.7 Métodos de coleções

- ▶ A classe **Collections** fornece vários algoritmos de alto desempenho para manipular elementos de coleção.
- ▶ Os algoritmos (Figura 20.5) são implementados como métodos **static**.



COMO PROGRAMAR

8^a edição

Método	Descrição
sort	Classifica os elementos de uma List.
binarySearch	Localiza um objeto em uma List.
reverse	Inverte os elementos de uma List.
shuffle	Ordena aleatoriamente os elementos de uma List.
fill	Configura todo elemento List para referir-se a um objeto especificado.
copy	Copia referências de uma List em outra.
min	Retorna o menor elemento em uma Collection.
max	Retorna o maior elemento em uma Collection.
addAll	Acrescenta todos os elementos em um array a uma Collection.
frequency	Calcula quantos elementos da coleção são iguais ao elemento especificado.
disjoint	Determina se duas coleções não têm nenhum elemento em comum.

Figura 20.5 | Métodos Collections.



COMO PROGRAMAR

8^a edição



Observação de engenharia de software 20.4

Os métodos da estrutura de coleções são polimórficos. Isto é, cada um deles pode operar em objetos que implementam interfaces específicas, independentemente da implementação subjacente.



COMO PROGRAMAR

8^a edição

20.7.1 Método sort

- ▶ O método **sort** classifica os elementos de uma **List**.
- ▶ Os elementos devem implementar a interface **Comparable**.
- ▶ A ordem é determinada pela ordem natural do tipo dos elementos como implementado por um método **compareTo**.
- ▶ O método **compareTo** é declarado na interface **Comparable** e, às vezes, é chamado de **método natural de comparação**.
- ▶ A chamada **sort** pode especificar como um segundo argumento um objeto **Comparator** que determina uma ordem alternativa dos elementos.



COMO PROGRAMAR

8^a edição

```
1 // Figura 20.6: Sort1.java
2 // Método Collections sort.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 public class Sort1
8 {
9     public static void main( String[] args )
10    {
11        String[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };
12
13        // Cria e exibe uma lista que contém os elementos do array de ternos
14        List< String > list = Arrays.asList( suits ); // cria List
15        System.out.printf( "Unsorted array elements: %s\n", list );
16
17        Collections.sort( list ); // classifica ArrayList ← Elementos list devem ser
18                                Comparable
18
19        // gera a saída da lista
20        System.out.printf( "Sorted array elements: %s\n", list );
21    } // fim de main
22 } // fim da classe Sort1
```

Figura 20.6 | Método Collections sort. (Parte 1 de 2.)



COMO PROGRAMAR

8^a edição

Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]

Sorted array elements: [Clubs, Diamonds, Hearts, Spades]

Figura 20.6 | Método Collections.sort. (Parte 2 de 2.)



COMO PROGRAMAR

8^a edição

- ▶ A interface **Comparator** é utilizada para classificar elementos de uma **Collection** em uma ordem diferente.
- ▶ O método **static Collections reverseOrder** retorna um objeto **Comparator** que ordena os elementos da coleção na ordem inversa.



COMO PROGRAMAR

8^a edição

```
1 // Figura 20.7: Sort2.java
2 // Utilizando um objeto Comparator com o método sort.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 public class Sort2
8 {
9     public static void main( String[] args )
10    {
11        String[] suits = { "Hearts", "Diamonds", "Clubs", "Spades" };
12
13        // Cria e exibe uma lista que contém os elementos do array de ternos
14        List< String > list = Arrays.asList( suits ); // cria List
15        System.out.printf( "Unsorted array elements: %s\n", list );
16
17        // classifica em ordem decrescente utilizando um comparador
18        Collections.sort( list, Collections.reverseOrder() );
19
20        // gera a saída de elementos List
21        System.out.printf( "Sorted list elements: %s\n", list );
22    } // fim de main
23 } // fim da classe Sort2
```

Comparator inverte a
ordem de classificação

Figura 20.7 | O método Collections sort com um objeto Comparator. (Parte I de 2.)



COMO PROGRAMAR

8^a edição

Unsorted array elements: [Hearts, Diamonds, Clubs, Spades]

Sorted list elements: [Spades, Hearts, Diamonds, Clubs]

Figura 20.7 | O método Collections.sort com um objeto Comparator. (Parte 2 de 2.)



COMO PROGRAMAR

8^a edição

- ▶ A Figura 20.8 cria uma classe **Comparator** personalizada, chamada **TimeComparator**, que implementa a interface **Comparator** para comparar dois objetos **Time2**.
- ▶ A classe **Time2**, declarada na Figura 8.5, representa o tempo em horas, minutos e segundos.
- ▶ A classe **TimeComparator** implementa a interface **Comparator**, um tipo genérico que aceita argumento de um único tipo.
- ▶ Uma classe que implementa **Comparator** deve declarar um método **compare** que recebe dois argumentos e retorna um número inteiro negativo se o primeiro argumento for menor que o segundo, 0 se os argumentos forem iguais ou um número inteiro positivo se o primeiro argumento for maior do que o segundo.



COMO PROGRAMAR

8^a edição

```
1 // Figura 20.8: TimeComparator.java
2 // Classe Comparator personalizada que compara dois objetos Time2.
3 import java.util.Comparator;
4
5 public class TimeComparator implements Comparator< Time2 > ← Comparator personalizado
6 {                                         para objetos Time2
7     public int compare(Time2 time1, Time2 time2 )
8     {
9         int hourCompare = time1.getHour() - time2.getHour(); // compara hora
10
11        // testa a primeira hora
12        if ( hourCompare != 0 )
13            return hourCompare;
14
15        int minuteCompare =
16            time1.getMinute() - time2.getMinute(); // compara minuto
17
18        // então testa o minuto
19        if ( minuteCompare != 0 )
20            return minuteCompare;
21
22        int secondCompare =
23            time1.getSecond() - time2.getSecond(); // compara segundo
```

Figura 20.8 | Classe Comparator personalizada que compara dois objetos Time2. (Parte 1 de 2.)



COMO PROGRAMAR

8^a edição

```
24
25     return secondCompare; // retorna o resultado da comparação de segundos
26 } // fim do método compare
27 } // fim da classe TimeComparator
```

Figura 20.8 | Classe Comparator personalizada que compara dois objetos Time2. (Parte 2 de 2.)



COMO PROGRAMAR

8^a edição

```
1 // Figura 20.9: Sort3.java
2 // Método Collections sort com um objeto Comparator personalizado.
3 import java.util.List;
4 import java.util.ArrayList;
5 import java.util.Collections;
6
7 public class Sort3
8 {
9     public static void main( String[] args )
10    {
11        List< Time2 > list = new ArrayList< Time2 >(); // cria List
12
13        list.add( new Time2( 6, 24, 34 ) );
14        list.add( new Time2( 18, 14, 58 ) );
15        list.add( new Time2( 6, 05, 34 ) );
16        list.add( new Time2( 12, 14, 58 ) );
17        list.add( new Time2( 6, 24, 22 ) );
18
19        // gera a saída de elementos List
20        System.out.printf( "Unsorted array elements:\n%s\n", list );
21    }
```

Figura 20.9 | Método Collections sort com um objeto Comparator personalizado. (Parte 1 de 2.)



COMO PROGRAMAR

8^a edição

```
22     // classifica em ordem utilizando um comparador
23     Collections.sort( list, new TimeComparator() );
24
25     // gera a saída de elementos List
26     System.out.printf( "Sorted list elements:\n%s\n", list );
27 } // fim de main
28 } // fim da classe Sort3
```

Objetos Time2 não podiam ser classificados antes de se criar o TimeComparator; a técnica pode ser utilizada para tornar objetos de praticamente qualquer classe classificáveis

```
Unsorted array elements:
[6:24:34 AM, 6:14:58 PM, 6:05:34 AM, 12:14:58 PM, 6:24:22 AM]
Sorted list elements:
[6:05:34 AM, 6:24:22 AM, 6:24:34 AM, 12:14:58 PM, 6:14:58 PM]
```

Figura 20.9 | Método Collections sort com um objeto Comparator personalizado. (Parte 2 de 2.)



COMO PROGRAMAR

8^a edição

20.7.2 Método shuffle

- ▶ O método **shuffle** ordena aleatoriamente os elementos de uma `List`.



COMO PROGRAMAR

8^a edição

```
1 // Figura 20.10: DeckOfCards.java
2 // Embaralhando e distribuindo cartas com o método Collections shuffle.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 // classe para representar uma carta de um baralho
8 class Card
9 {
10     public static enum Face { Ace, Deuce, Three, Four, Five, Six,
11         Seven, Eight, Nine, Ten, Jack, Queen, King };
12     public static enum Suit { Clubs, Diamonds, Hearts, Spades };
13
14     private final Face face; // face da carta
15     private final Suit suit; // naipe da carta
16
17     // construtor de dois argumentos
18     public Card( Face cardFace, Suit cardSuit )
19     {
20         face = cardFace; // inicializa face da carta
21         suit = cardSuit; // inicializa naipe da carta
22     } // fim do construtor Card de dois argumentos
```

Figura 20.10 | Embaralhamento e distribuição de cartas com o método Collections shuffle.
(Parte 1 de 5.)



COMO PROGRAMAR

8^a edição

```
23
24     // retorna a face da carta
25     public Face getFace()
26     {
27         return face;
28     } // fim do método getFace
29
30     // retorna o naipe da carta
31     public Suit getSuit()
32     {
33         return suit;
34     } // fim do método getSuit
35
36     // retorna a representação String de Card
37     public String toString()
38     {
39         return String.format( "%s of %s", face, suit );
40     } // fim do método toString
41 } // fim da classe Card
42
```

Figura 20.10 | Embaralhamento e distribuição de cartas com o método Collections shuffle.
(Parte 2 de 5.)



COMO PROGRAMAR

8^a edição

```
43 // declaração da classe DeckOfCards
44 public class DeckOfCards
45 {
46     private List< Card > list; // declara List que armazenará cartas
47
48     // configura o baralho de cartas e embaralha
49     public DeckOfCards()
50     {
51         Card[] deck = new Card[ 52 ];
52         int count = 0; // número de cartas
53
54         // preenche baralho com objetos Card
55         for (Card.Suit suit : Card.Suit.values() )
56         {
57             for (Card.Face face : Card.Face.values() )
58             {
59                 deck[ count ] = new Card( face, suit );
60                 ++count;
61             } // for final
62         } // for final
63     }
```

Figura 20.10 | Embaralhamento e distribuição de cartas com o método Collections shuffle.
(Parte 3 de 5.)

COMO PROGRAMAR

8ª edição

```
64     list = Arrays.asList( deck ); // obtém List
65     Collections.shuffle( list ); // embaralha as cartas
66 } // fim do construtor DeckOfCards
67
68 // gera a saída de baralho
69 public void printCards()
70 {
71     // exibe 52 cartas em duas colunas
72     for ( int i = 0; i < list.size(); i++ )
73         System.out.printf( "%-19s%s",
74             list.get( i ),
75             ( ( i + 1 ) % 4 == 0 ) ? "\n" : "" );
76 } // fim do método printCards
77
78 public static void main( String[] args )
79 {
80     DeckOfCards cards = new DeckOfCards();
81     cards.printCards();
82 } // fim de main
83 } // fim da classe DeckOfCards
```

Embaralha o conteúdo
de uma coleção

Figura 20.10 | Embaralhamento e distribuição de cartas com o método `Collections shuffle`.
(Parte 4 de 5.)



COMO PROGRAMAR

8^a edição

Deuce of Clubs	Six of Spades	Nine of Diamonds	Ten of Hearts
Three of Diamonds	Five of Clubs	Deuce of Diamonds	Seven of Clubs
Three of Spades	Six of Diamonds	King of Clubs	Jack of Hearts
Ten of Spades	King of Diamonds	Eight of Spades	Six of Hearts
Nine of Clubs	Ten of Diamonds	Eight of Diamonds	Eight of Hearts
Ten of Clubs	Five of Hearts	Ace of Clubs	Deuce of Hearts
Queen of Diamonds	Ace of Diamonds	Four of Clubs	Nine of Hearts
Ace of Spades	Deuce of Spades	Ace of Hearts	Jack of Diamonds
Seven of Diamonds	Three of Hearts	Four of Spades	Four of Diamonds
Seven of Spades	King of Hearts	Seven of Hearts	Five of Diamonds
Eight of Clubs	Three of Clubs	Queen of Clubs	Queen of Spades
Six of Clubs	Nine of Spades	Four of Hearts	Jack of Clubs
Five of Spades	King of Spades	Jack of Spades	Queen of Hearts

Figura 20.10 | Embaralhamento e distribuição de cartas com o método Collections shuffle.
(Parte 5 de 5.)



COMO PROGRAMAR

8^a edição

20.7.3 Métodos `reverse`, `fill`, `copy`, `max` e `min`

- ▶ O método **Collections reverse** inverte a ordem dos elementos em uma `List`.
- ▶ O método **fill** sobrescreve elementos em uma `List` com um valor específico.
- ▶ O método **copy** recebe dois argumentos — uma `List` de destino e uma `List` de origem.
- ▶ Cada elemento da `List` de origem é copiado para a `List` de destino.
- ▶ A `List` de destino deve ser pelo menos tão longa quanto a `List` de origem; caso contrário, uma `IndexOutOfBoundsException` ocorre.
- ▶ Se a `List` de destino for mais longa, os elementos não sobrescritos permanecerão inalterados.
- ▶ Os métodos **min** e **max** operam sobre qualquer `Collection`.
- ▶ O método **min** retorna o menor elemento de uma `Collection` e o método **max** retorna o maior elemento de uma `Collection`.



COMO PROGRAMAR

8^a edição

```
1 // Figura 20.11: Algorithms1.java
2 // Métodos Collections reverse, fill, copy, max e min.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 public class Algorithms1
8 {
9     public static void main( String[] args )
10    {
11        // cria e exibe uma List< Character >
12        Character[] letters = { 'P', 'C', 'M' };
13        List< Character > list = Arrays.asList( letters ); // obtém List
14        System.out.println( "list contains: " );
15        output( list );
16
17        // inverte e exibe a List< Character >
18        Collections.reverse( list ); // inverte a ordem dos elementos
19        System.out.println( "\nAfter calling reverse, list contains: " );
20        output( list );
21    }
```

Figura 20.11 | Os métodos Collections reverse, fill, copy, max e min. (Parte I de 3.)



COMO PROGRAMAR

8^a edição

```
22 // cria copyList de um array de 3 caracteres
23 Character[] lettersCopy = new Character[ 3 ];
24 List< Character > copyList = Arrays.asList( lettersCopy );
25
26 // copia os conteúdos de list para copyList
27 Collections.copy( copyList, list );
28 System.out.println( "\nAfter copying, copyList contains: " );
29 output( copyList );
30
31 // preenche a lista com Rs
32 Collections.fill( list, 'R' );
33 System.out.println( "\nAfter calling fill, list contains: " );
34 output( list );
35 } // fim de main
36
37 // envia informações de List para saída
38 private static void output( List< Character > listRef )
39 {
40     System.out.print( "The list is: " );
41
42     for ( Character element : listRef )
43         System.out.printf( "%s ", element );
```

Figura 20.11 | Os métodos Collections reverse, fill, copy, max e min. (Parte 2 de 3.)



COMO PROGRAMAR

8^a edição

```
44
45     System.out.printf( "\nMax: %s", Collections.max( listRef ) );
46     System.out.printf( " Min: %s\n", Collections.min( listRef ) );
47 } // fim do método output
48 } // fim da classe Algorithms1
```

```
list contains:
The list is: P C M
Max: P Min: C
```

```
After calling reverse, list contains:
The list is: M C P
Max: P Min: C
```

```
After copying, copyList contains:
The list is: M C P
Max: P Min: C
```

```
After calling fill, list contains:
The list is: R R R
Max: R Min: R
```

Figura 20.11 | Os métodos Collections reverse, fill, copy, max e min. (Parte 3 de 3.)



COMO PROGRAMAR

8^a edição

20.7.4 Método `binarySearch`

- ▶ O método `static Collections binarySearch` localiza um objeto em uma `List`.
- ▶ Se o objeto for encontrado, seu índice será retornado.
- ▶ Se o objeto não for localizado, `binarySearch` retornará um valor negativo.
- ▶ O método `binarySearch` determina esse valor negativo primeiro calculando o ponto de inserção e tornando seu sinal negativo.
- ▶ Então, `binarySearch` subtrai 1 do ponto de inserção para obter o valor de retorno, que garante que o método `binarySearch` retorna números positivos ($>=0$) se e somente se o objeto for localizado.



COMO PROGRAMAR

8^a edição

```
1 // Figura 20.12: BinarySearchTest.java
2 // Método Collections binarySearch.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6 import java.util.ArrayList;
7
8 public class BinarySearchTest
9 {
10     public static void main( String[] args )
11     {
12         // cria um ArrayList< String > do conteúdo do array colors
13         String[] colors = { "red", "white", "blue", "black", "yellow",
14                           "purple", "tan", "pink" };
15         List< String > list =
16             new ArrayList< String >( Arrays.asList( colors ) );
17
18         Collections.sort( list ); // classifica a ArrayList
19         System.out.printf( "Sorted ArrayList: %s\n", list );
20
21         // pesquisa vários valores na lista
22         printSearchResults( list, colors[ 3 ] ); // primeiro item
23         printSearchResults( list, colors[ 0 ] ); // item do meio
```

Figura 20.12 | Método Collections binarySearch. (Parte I de 3.)



COMO PROGRAMAR

8^a edição

```
24     printSearchResults( list, colors[ 7 ] ); // último item
25     printSearchResults( list, "aqua" ); // abaixo do mais baixo
26     printSearchResults( list, "gray" ); // não existe
27     printSearchResults( list, "teal" ); // não existe
28 } // fim de main
29
30 // realiza pesquisa e exibe o resultado
31 private static void printSearchResults(
32     List< String > list, String key )
33 {
34     int result = 0;
35
36     System.out.printf( "\nSearching for: %s\n", key );
37     result = Collections.binarySearch( list, key ); ← A coleção deve ser primeiro
38                                         classificada
39     if ( result >= 0 )
40         System.out.printf( "Found at index %d\n", result );
41     else
42         System.out.printf( "Not Found (%d)\n", result );
43 } // fim do método printSearchResults
44 } // fim da classe BinarySearchTest
```

Figura 20.12 | Método Collections binarySearch. (Parte 2 de 3.)



COMO PROGRAMAR

8^a edição

```
Sorted ArrayList: [black, blue, pink, purple, red, tan, white, yellow]
```

```
Searching for: black
Found at index 0
```

```
Searching for: red
Found at index 4
```

```
Searching for: pink
Found at index 2
```

```
Searching for: aqua
Not Found (-1)
```

```
Searching for: gray
Not Found (-3)
```

```
Searching for: teal
Not Found (-7)
```

Figura 20.12 | Método Collections.binarySearch. (Parte 3 de 3.)



COMO PROGRAMAR

8^a edição

20.7.5 Métodos addAll, frequency e disjoint

- ▶ O método **Collections addAll** aceita dois argumentos — uma **Collection** na qual inserir o(s) novo(s) elemento(s) e um array que fornece elementos a ser inseridos.
- ▶ O método **Collections frequency** aceita dois argumentos — uma **Collection** a ser pesquisada e um **Object** a ser procurado na coleção.
- ▶ O método **frequency** retorna o número de vezes que o segundo argumento aparece na coleção.
- ▶ O método **Collections disjoint** aceita duas **Collections** e retorna **true** se elas não tiverem nenhum elemento em comum.



COMO PROGRAMAR

8^a edição

```
1 // Figura 20.13: Algorithms2.java
2 // Métodos Collections.addAll, frequency e disjoint.
3 import java.util.ArrayList;
4 import java.util.List;
5 import java.util.Arrays;
6 import java.util.Collections;
7
8 public class Algorithms2
9 {
10     public static void main( String[] args )
11     {
12         // inicializa list1 e list2
13         String[] colors = { "red", "white", "yellow", "blue" };
14         List< String > list1 = Arrays.asList( colors );
15         ArrayList< String > list2 = new ArrayList< String >();
16
17         list2.add( "black" ); // adiciona "black" ao fim de list2
18         list2.add( "red" ); // adiciona "red" ao fim de list2
19         list2.add( "green" ); // adiciona "green" ao fim de list2
20
21         System.out.print( "Before addAll, list2 contains: " );
22 }
```

Figura 20.13 | Métodos Collections.addAll, frequency e disjoint. (Parte I de 3.)



COMO PROGRAMAR

8^a edição

```
23     // exibe elementos em list2
24     for ( String s : list2 )
25         System.out.printf( "%s ", s );
26
27     Collections.addAll( list2, colors ); // adiciona Strings colors a list2
28
29     System.out.print( "\nAfter addAll, list2 contains: " );
30
31     // exibe elementos em list2
32     for ( String s : list2 )
33         System.out.printf( "%s ", s );
34
35     // obtém frequência de "red"
36     int frequency = Collections.frequency( list2, "red" );
37     System.out.printf(
38         "\nFrequency of red in list2: %d\n", frequency );
39
40     // verifica se list1 e list2 têm elementos em comum
41     boolean disjoint = Collections.disjoint( list1, list2 );
42
43     System.out.printf( "list1 and list2 %s elements in common\n",
44         ( disjoint ? "do not have" : "have" ) );
45 }
46 } // fim da classe Algorithms2
```

Figura 20.13 | Métodos Collections addAll, frequency e disjoint. (Parte 2 de 3.)



COMO PROGRAMAR

8^a edição

Before addAll, list2 contains: black red green

After addAll, list2 contains: black red green red white yellow blue

Frequency of red in list2: 2

list1 and list2 have elements in common

Figura 20.13 | Métodos Collections addAll, frequency e disjoint. (Parte 3 de 3.)



COMO PROGRAMAR

8^a edição

20.8 Classe Stack do pacote java.util

- ▶ A classe **Stack** do pacote de utilitários Java (`java.util`) estende a classe **Vector** para implementar uma estrutura de dados de pilha.
- ▶ O método **Stack push** adiciona um objeto **Number** à parte superior da pilha.
- ▶ Qualquer literal inteiro que tenha o **sufixo L** é um valor `long`.
- ▶ Um literal de inteiro sem sufixo é um valor `int`.
- ▶ Qualquer literal de ponto flutuante que tenha o **sufixo F** é um valor `float`.
- ▶ Um literal de ponto flutuante sem sufixo é um valor `double`.
- ▶ O método **Stack pop** remove o elemento superior da pilha.
- ▶ Se não houver nenhum elemento na **Stack**, o método **pop** lança uma **EmptyStackException**, que termina o loop.
- ▶ O método **peek** retorna o elemento no topo da pilha sem remover o elemento da pilha.
- ▶ O método **isEmpty** determina se a pilha está vazia.



COMO PROGRAMAR

8^a edição



Dica de prevenção de erro 20.1

Como `Stack` estende `Vector`, todos os métodos `public Vector` podem ser chamados em objetos `Stack`, mesmo se os métodos não representarem operações de pilha convencionais. Por exemplo, o método `Vector add` pode ser utilizado para inserir um elemento em qualquer lugar em uma pilha — uma operação que poderia “corromper” a pilha. Ao manipular uma `Stack`, somente os métodos `push` e `pop` devem ser utilizados para adicionar elementos à `Stack` e remover elementos dela, respectivamente.



COMO PROGRAMAR

8^a edição

```
1 // Figura 20.14: Stacktest.java
2 // Classe Stack do pacote java.util.
3 import java.util.Stack;
4 import java.util.EmptyStackException;
5
6 public class StackTest
7 {
8     public static void main( String[] args )
9     {
10         Stack< Number > stack = new Stack< Number >(); // cria uma pilha
11
12         // utiliza método push
13         stack.push( 12L ); // adiciona valor long 12L
14         System.out.println( "Pushed 12L" );
15         printStack( stack );
16         stack.push( 34567 ); // adiciona valor int 34567
17         System.out.println( "Pushed 34567" );
18         printStack( stack );
19         stack.push( 1.0F ); // adiciona valor float 1.0F
20         System.out.println( "Pushed 1.0F" );
21         printStack( stack );
22         stack.push( 1234.5678 ); // adiciona valor double 1234.5678
23         System.out.println( "Pushed 1234.5678 " );
24         printStack( stack );
```

Figura 20.14 | Classe Stack do pacote java.util. (Parte 1 de 4.)



COMO PROGRAMAR

8^a edição

```
25
26     // remove itens de pilha
27     try
28     {
29         Number removedObject = null;
30
31         // remove elementos da pilha
32         while ( true )
33         {
34             removedObject = stack.pop(); // utiliza método pop
35             System.out.printf( "Popped %s\n", removedObject );
36             printStack( stack );
37         } // fim do while
38     } // fim do try
39     catch ( EmptyStackException emptyStackException )
40     {
41         emptyStackException.printStackTrace();
42     } // fim do catch
43 } // fim de main
44
45     // exibe o conteúdo de Stack
46     private static void printStack( Stack< Number > stack )
47 {
```

Figura 20.14 | Classe Stack do pacote java.util. (Parte 2 de 4.)



COMO PROGRAMAR

8^a edição

```
48     if (stack.isEmpty() )
49         System.out.println( "stack is empty\n" ); // a pilha está vazia
50     else // a pilha não está vazia
51         System.out.printf( "stack contains: %s (top)\n", stack );
52     } // fim do método printStack
53 } // fim da classe StackTest
```

Figura 20.14 | Classe Stack do pacote java.util. (Parte 3 de 4.)



COMO PROGRAMAR

8^a edição

```
Pushed 12L
stack contains: [12] (top)
Pushed 34567
stack contains: [12, 34567] (top)
Pushed 1.0F
stack contains: [12, 34567, 1.0] (top)
Pushed 1234.5678
stack contains: [12, 34567, 1.0, 1234.5678] (top)
Popped 1234.5678
stack contains: [12, 34567, 1.0] (top)
Popped 1.0
stack contains: [12, 34567] (top)
Popped 34567
stack contains: [12] (top)
Popped 12
stack is empty

java.util.EmptyStackException
    at java.util.Stack.peek(Unknown Source)
    at java.util.Stack.pop(Unknown Source)
    at StackTest.main(StackTest.java:34)
```

Figura 20.14 | Classe Stack do pacote `java.util`. (Parte 4 de 4.)



COMO PROGRAMAR

8^a edição

20.9 Classe PriorityQueue e interface Queue

- ▶ A interface **Queue** estende a interface **Collection** e fornece operações adicionais para inserir, remover e inspecionar elementos em uma fila.
- ▶ **PriorityQueue** ordena os elementos de acordo com seu ordenamento natural.
- ▶ Os elementos são inseridos na ordem de prioridade de tal modo que o elemento de maior prioridade (isto é, o maior valor) será o primeiro elemento removido da **PriorityQueue**.
- ▶ As operações **PriorityQueue** comuns são:
 - **offer** para inserir um elemento na localização adequada com base na ordem de prioridade.
 - **poll** para remover o elemento de mais alta prioridade da fila de prioridade.
 - **peek** para obter uma referência ao elemento de prioridade mais alta da fila de prioridades.
 - **clear** para remover todos os elementos da fila de prioridades.
 - **size** para obter o número de elementos na fila.



COMO PROGRAMAR

8^a edição

```
1 // Figura 20.15: PriorityQueueTest.java
2 // Programa de teste PriorityQueue.
3 import java.util.PriorityQueue;
4
5 public class PriorityQueueTest
6 {
7     public static void main( String[] args )
8     {
9         // fila de capacidade 11
10        PriorityQueue< Double > queue = new PriorityQueue< Double >(); ← A ordem natural determina
11
12        // insere elementos na fila
13        queue.offer( 3.2 );
14        queue.offer( 9.8 );
15        queue.offer( 5.4 );
16
17        System.out.print( "Polling from queue: " );
18    }
}
```

A ordem natural determina
a prioridade

Figura 20.15 | Programa de teste PriorityQueue. (Parte 1 de 2.)



COMO PROGRAMAR

8^a edição

```
19      // exibe elementos na fila
20      while (queue.size() > 0 )
21      {
22          System.out.printf( "%.1f ", queue.peek() ); // visualiza elemento superior
23          queue.poll(); // remove elemento superior
24      } // fim do while
25  } // fim de main
26 } // fim da classe PriorityQueueTest
```

```
Polling from queue: 3.2 5.4 9.8
```

Figura 20.15 | Programa de teste PriorityQueue. (Parte 2 de 2.)



COMO PROGRAMAR

8^a edição

20.10 Conjuntos

- ▶ Um **Set** é uma **Collection** não ordenada de elementos únicos (isto é, sem elementos duplicados).
- ▶ A estrutura de coleções contém diversas implementações de **Set**, incluindo **HashSet** e **TreeSet**.
- ▶ **HashSet** armazena seus elementos em uma tabela de hash e **TreeSet** armazena seus elementos em uma árvore.



COMO PROGRAMAR

8^a edição

```
1 // Figura 20.16: SetTest.java
2 // HashSet utilizado para remover valores duplicados do array de strings.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.HashSet;
6 import java.util.Set;
7 import java.util.Collection;
8
9 public class SetTest
10 {
11     public static void main( String[] args )
12     {
13         // cria e exibe uma List< String >
14         String[] colors = { "red", "white", "blue", "green", "gray",
15             "orange", "tan", "white", "cyan", "peach", "gray", "orange" };
16         List< String > list = Arrays.asList( colors );
17         System.out.printf( "List: %s\n", list );
18
19         // elimina duplicatas e imprime valores únicos
20         printNonDuplicates( list );
21     } // fim de main
22 }
```

Figura 20.16 | HashSet utilizado para remover valores duplicados de um array de strings. (Parte I de 2.)



COMO PROGRAMAR

8^a edição

```
23     // cria um Set a partir de uma Collection para eliminar duplicatas
24     private static void printNonDuplicates( Collection< String > values )
25     {
26         // cria um HashSet
27         Set< String > set = new HashSet< String >( values );
28
29         System.out.print( "\nNonduplicates are: " );
30
31         for ( String value : set )
32             System.out.printf( "%s ", value );
33
34         System.out.println();
35     } // fim do método printNonDuplicates
36 } // fim da classe SetTest
```

List: [red, white, blue, green, gray, orange, tan, white, cyan, peach, gray, orange]

Nonduplicates are: orange green white peach gray cyan red blue tan

Figura 20.16 | HashSet utilizado para remover valores duplicados de um array de strings. (Parte 2 de 2.)



COMO PROGRAMAR

8^a edição

- ▶ A estrutura de coleções também inclui a interface **SortedSet** (que estende **Set**) para conjuntos que mantêm seus elementos em ordem de classificação.
- ▶ A classe **TreeSet** implementa **SortedSet**.
- ▶ O método **TreeSet headSet** obtém um subconjunto de **TreeSet** em que todo elemento é menor do que o valor especificado.
- ▶ O método **TreeSet tailSet** obtém um subconjunto em que cada elemento é maior ou igual a um elemento especificado.
- ▶ Os métodos **SortedSet first** e **last** obtêm os menores e os maiores elementos do conjunto, respectivamente.



COMO PROGRAMAR

8^a edição

```
1 // Figura 20.17: SortedSetTest.java
2 // Utilizando SortedSets e TreeSets.
3 import java.util.Arrays;
4 import java.util.SortedSet;
5 import java.util.TreeSet;
6
7 public class SortedSetTest
8 {
9     public static void main( String[] args )
10    {
11        // cria TreeSet a partir do array colors
12        String[] colors = { "yellow", "green", "black", "tan", "grey",
13                           "white", "orange", "red", "green" };
14        SortedSet< String > tree =
15            new TreeSet< String >( Arrays.asList( colors ) );
16
17        System.out.print( "sorted set: " );
18        printSet( tree ); // conteúdo de saída de árvore
19
20        // obtém headSet com base em "orange"
21        System.out.print( "headSet (\"orange\"): " );
22        printSet( tree.headSet( "orange" ) );
23    }
```

Figura 20.17 | Utilizando SortedSets e TreeSets. (Parte 1 de 3.)



COMO PROGRAMAR

8^a edição

```
24      // obtém tailSet baseado em "orange"
25      System.out.print( "tailSet (\"orange\"): " );
26      printSet(tree.tailSet( "orange" ) );
27
28      // obtém o primeiro e o último elementos
29      System.out.printf( "first: %s\n", tree.first() );
30      System.out.printf( "last : %s\n", tree.last() );
31 } // fim de main
32
33 // imprime a SortedSet utilizando a instrução for aprimorada
34 private static void printSet( SortedSet< String > set )
35 {
36     for ( String s : set )
37         System.out.printf( "%s ", s );
38
39     System.out.println();
40 } // fim do método printSet
41 } // fim da classe SortedSetTest
```

Figura 20.17 | Utilizando SortedSets e TreeSetes. (Parte 2 de 3.)



COMO PROGRAMAR

8^a edição

```
sorted set: black green grey orange red tan white yellow
headSet ("orange"): black green grey
tailSet ("orange"): orange red tan white yellow
first: black
last : yellow
```

Figura 20.17 | Utilizando SortedSets e TreeSet. (Parte 3 de 3.)



COMO PROGRAMAR

8^a edição

20.11 Mapas

- ▶ **Maps** associam chaves a valores.
- ▶ As chaves de um **Map** devem ser únicas, mas os valores associados, não.
- ▶ Se um **Map** contiver tanto chaves como valores únicos, dizemos que ele implementa um **mapeamento um para um**.
- ▶ Se apenas as chaves forem únicas, dizemos que o **Map** implementa um **mapeamento de muitos para muitos** — muitas chaves podem mapear para um valor.
- ▶ Três das várias classes que implementam a interface **Map** são **Hashtable**, **HashMap** e **TreeMap**.
- ▶ **Hashtables** e **HashMaps** armazenam elementos em tabelas hash, e **TreeMaps** armazenam elementos em árvores.



COMO PROGRAMAR

8^a edição

- ▶ A interface **SortedMap** estende **Map** e mantém suas chaves em ordem de classificação — na ordem natural dos elementos ou em uma ordem especificada por um **Comparator**.
- ▶ A classe **TreeMap** implementa **SortedMap**.
- ▶ Hashing é um esquema de alta velocidade para converter chaves em índices de arrays únicos.
- ▶ O **fator de carga** de uma tabela de hash afeta o desempenho de esquemas de hashing.
- ▶ O fator de carga é a relação do número de células ocupadas na tabela de hash com o número total de células na tabela de hash.
- ▶ Quanto mais a proporção se aproxima de 1,0, maior a chance de colisões.



COMO PROGRAMAR

8^a edição



Dica de desempenho 20.2

O fator de carga em uma tabela de hash é um exemplo clássico de uma troca entre espaço de memória e tempo de execução: aumentando o fator de carga, melhoramos a utilização da memória, mas o programa executa mais lentamente por causa do aumento das colisões de hashing. Diminuindo o fator de carga, melhoramos a velocidade do programa, em virtude da redução das colisões de hashing, mas fazemos uma pobre utilização da memória porque uma parte maior da tabela de hash permanece vazia.



COMO PROGRAMAR

8^a edição

```
1 // Figura 20.18: WordTypeCount.java
2 // O programa conta o número de ocorrências de cada palavra em uma String.
3 import java.util.Map;
4 import java.util.HashMap;
5 import java.util.Set;
6 import java.util.TreeSet;
7 import java.util.Scanner;
8
9 public class WordTypeCount
10 {
11     public static void main( String[] args )
12     {
13         // cria HashMap para armazenar chaves de String e valores Integer
14         Map< String, Integer > myMap = new HashMap< String, Integer >();
15
16         createMap( myMap ); // cria mapa com base na entrada do usuário
17         displayMap( myMap ); // exibe o conteúdo do mapa
18     } // fim de main
19
20     // cria mapa de entrada do usuário
21     private static void createMap( Map< String, Integer > map )
22     {
```

Figura 20.18 | O programa conta o número de ocorrências de cada palavra em uma String. (Parte 1 de 4.)



COMO PROGRAMAR

8^a edição

```
23     Scanner scanner = new Scanner( System.in ); // cria o scanner
24     System.out.println( "Enter a string:" ); // solicita a entrada do usuário
25     String input = scanner.nextLine();
26
27     // tokeniza a entrada
28     String[] tokens = input.split( " " );
29
30     // processamento de texto de entrada
31     for ( String token : tokens )
32     {
33         String word = token.toLowerCase(); // obtém palavra em letras minúsculas
34
35         // se o mapa contiver a palavra
36         if (map.containsKey( word ) ) // is word in map
37         {
38             int count = map.get( word ); // obtém a contagem atual
39             map.put( word, count + 1 ); // incrementa a contagem
40         } // fim do if
41         else
42             map.put( word, 1 ); // adiciona nova palavra com uma contagem de 1 para mapa
43     } // fim do for
44 } // fim do método createMap
```

Figura 20.18 | O programa conta o número de ocorrências de cada palavra em uma `String`. (Parte 2 de 4.)



COMO PROGRAMAR

8^a edição

```
45
46     // exibe o conteúdo do mapa
47     private static void displayMap( Map< String, Integer > map )
48     {
49         Set< String > keys = map.keySet(); //obtém chaves
50
51         // classifica as chaves
52         TreeSet< String > sortedKeys = new TreeSet< String >( keys );
53
54         System.out.println( "\nMap contains:\nKey\t\tValue" );
55
56         // gera a saída de cada chave no mapa
57         for ( String key : sortedKeys )
58             System.out.printf( "%-10s%10s\n", key, map.get( key ) );
59
60         System.out.printf(
61             "\nsize: %d\nisEmpty: %b\n", map.size(), map.isEmpty() );
62     } // fim do método displayMap
63 } // fim da classe WordTypeCount
```

Figura 20.18 | O programa conta o número de ocorrências de cada palavra em uma String. (Parte 3 de 4.)



COMO PROGRAMAR

8^a edição

Enter a string:

this is a sample sentence with several words this is another sample sentence with several different words

Map contains:

Key	Value
a	1
another	1
different	1
is	2
sample	2
sentence	2
several	2
this	2
with	2
words	2

size: 10

isEmpty: false

Figura 20.18 | O programa conta o número de ocorrências de cada palavra em uma String. (Parte 4 de 4.)



COMO PROGRAMAR

8^a edição

- ▶ O método **Map containsKey** determina se a chave está em um mapa.
- ▶ O método **Map put** cria uma nova entrada no mapa ou substitui um valor de entrada existente.
- ▶ O método **put** retorna o valor anterior associado com a chave, ou **null** se a chave não estiver no mapa.
- ▶ O método **get** obtém o valor associado com a chave especificada no mapa.
- ▶ O método **HashMap keySet** retorna um conjunto de chaves.
- ▶ O método **size** retorna o número dos pares de chaves/valor no Map.
- ▶ O método **isEmpty** retorna um boolean que indica se o Map está vazio.



COMO PROGRAMAR

8^a edição

20.12 Classe Properties

- ▶ Um objeto **Properties** é um **Hashtable** persistente que normalmente armazena os pares de chave/valor de strings — supondo que você utiliza métodos **setProperty** e **getProperty** para manipular a tabela em vez dos métodos **Hashtable** **put** e **get** herdados.
- ▶ O conteúdo do objeto **Properties** pode ser gravado em um fluxo de saída (possivelmente um arquivo) e lido de volta por um fluxo de entrada.
- ▶ Um uso comum de objetos **Properties** em versões anteriores do Java era manter os dados de configuração de aplicativo ou preferências de usuário de aplicativos.
- ▶ [Nota: A **Preferences API** (pacote **java.util.prefs**) foi projetada para substituir esse uso da classe **Properties**.]



COMO PROGRAMAR

8^a edição

- ▶ O método **Properties store** salva o conteúdo do objeto no **OutputStream** especificado como o primeiro argumento. O segundo argumento, uma **String**, é uma descrição gravada no arquivo.
- ▶ O método **Properties list**, que aceita um argumento **PrintStream**, é útil para exibir a lista de propriedades.
- ▶ O método **Properties load** restaura o conteúdo de um objeto **Properties** a partir do **InputStream** especificado como o primeiro argumento (nesse caso, um **FileInputStream**).



COMO PROGRAMAR

8^a edição

```
1 // Figura 20.19: PropertiesTest.java
2 // Demonstra classe Properties do pacote java.util.
3 import java.io.FileOutputStream;
4 import java.io.FileInputStream;
5 import java.io.IOException;
6 import java.util.Properties;
7 import java.util.Set;
8
9 public class PropertiesTest
10 {
11     public static void main( String[] args )
12     {
13         Properties table = new Properties(); // cria tabela Properties
14
15         // configura propriedades
16         table.setProperty( "color", "blue" );
17         table.setProperty( "width", "200" );
18
19         System.out.println( "After setting properties" );
20         listProperties( table ); // exibe os valores da propriedade
21
22         // substitui o valor de propriedade
23         table.setProperty( "color", "red" );
```

Figura 20.19 | Classe Properties do pacote java.util. (Parte 1 de 5.)



COMO PROGRAMAR

8^a edição

```
24
25     System.out.println( "After replacing properties" );
26     listProperties( table ); // exibe os valores da propriedade
27
28     saveProperties( table ); // salva as propriedades
29
30     table.clear(); // tabela vazia
31
32     System.out.println( "After clearing properties" );
33     listProperties( table ); // exibe os valores da propriedade
34
35     loadProperties( table ); // carrega propriedades
36
37     // obtém valor de cor da propriedade
38     Object value = table.getProperty( "color" );
39
40     // verifica se o valor está na tabela
41     if ( value != null )
42         System.out.printf( "Property color's value is %s\n", value );
43     else
44         System.out.println( "Property color is not in table" );
45 } // fim de main
46
```

Figura 20.19 | Classe Properties do pacote java.util. (Parte 2 de 5.)



COMO PROGRAMAR

8^a edição

```
47     // salva as propriedades para um arquivo
48     private static void saveProperties( Properties props )
49     {
50         // salva o conteúdo da tabela
51         try
52         {
53             FileOutputStream output = new FileOutputStream( "props.dat" );
54             props.store( output, "Sample Properties" ); // salva as propriedades
55             output.close();
56             System.out.println( "After saving properties" );
57             listProperties( props ); // exibe os valores da propriedade
58         } // fim do try
59         catch ( IOException ioException )
60         {
61             ioException.printStackTrace();
62         } // fim do catch
63     } // fim do método saveProperties
64
65     // carrega as propriedades de um arquivo
66     private static void loadProperties( Properties props )
67     {
68         // carrega o conteúdo da tabela
69         try
70         {
```

Figura 20.19 | Classe Properties do pacote java.util. (Parte 3 de 5.)



COMO PROGRAMAR

8^a edição

```
71     FileInputStream input = new FileInputStream( "props.dat" );
72     props.load( input ); // carrega propriedades
73     input.close();
74     System.out.println( "After Loading properties" );
75     listProperties( props ); // exibe os valores da propriedade
76 } // fim do try
77 catch ( IOException ioException )
78 {
79     ioException.printStackTrace();
80 } // fim do catch
81 } // fim do método loadProperties
82
83 // gera saída de valores de propriedade
84 private static void listProperties( Properties props )
85 {
86     Set< Object > keys = props.keySet(); // obtém nomes de propriedade
87
88     // gera saída de pares nome/valor
89     for ( Object key : keys )
90         System.out.printf(
91             "%s\t%s\n", key, props.getProperty( ( String ) key ) );
92
93     System.out.println();
94 } // fim do método listProperties
95 } // fim da classe PropertiesTest
```

Figura 20.19 | Classe Properties do pacote java.util. (Parte 4 de 5.)



COMO PROGRAMAR

8^a edição

After setting properties

```
color    blue  
width   200
```

After replacing properties

```
color    red  
width   200
```

After saving properties

```
color    red  
width   200
```

After clearing properties

After loading properties

```
color    red  
width   200
```

Property color's value is red

Figura 20.19 | Classe Properties do pacote java.util. (Parte 5 de 5.)



COMO PROGRAMAR

8^a edição

20.13 Coleções sincronizadas

- ▶ Os **empacotadores de sincronização** são utilizados para coleções que podem ser acessadas por múltiplas threads.
- ▶ Um objeto **empacotador (wrapper)** recebe chamadas de método, adiciona sincronização de thread e delega as chamadas para o objeto de coleção empacotado.
- ▶ A API **Collections** fornece um conjunto de métodos **static** para empacotar coleções como versões sincronizadas.
- ▶ Os cabeçalhos de método para os empacotadores de sincronização são listados na Figura 20.20.



COMO PROGRAMAR

8^a edição

Cabeçalhos de método public static

```
< T > Collection< T > synchronizedCollection( Collection< T > c )
< T > List< T > synchronizedList( List< T > aList )
< T > Set< T > synchronizedSet( Set< T > s )
< T > SortedSet< T > synchronizedSortedSet( SortedSet< T > s )
< K, V > Map< K, V > synchronizedMap( Map< K, V > m )
< K, V > SortedMap< K, V > synchronizedSortedMap( SortedMap< K, V > m )
```

Figura 20.20 | Métodos empacotadores de sincronização.



COMO PROGRAMAR

8^a edição

20.14 Coleções não modificáveis

- ▶ A classe `Collections` fornece um conjunto de métodos `static` que criam **empacotadores não modificáveis** para coleções.
- ▶ Empacotadores não modificáveis lançam `UnsupportedOperationExceptions` se forem feitas tentativas de modificar a coleção.
- ▶ Os cabeçalhos para esses métodos são listados na Figura 20.21.



COMO PROGRAMAR

8^a edição

Cabeçalhos de método public static

```
< T > Collection< T > unmodifiableCollection( Collection< T > c )
< T > List< T > unmodifiableList( List< T > aList )
< T > Set< T > unmodifiableSet( Set< T > s )
< T > SortedSet< T > unmodifiableSortedSet( SortedSet< T > s )
< K, V > Map< K, V > unmodifiableMap( Map< K, V > m )
< K, V > SortedMap< K, V > unmodifiableSortedMap( SortedMap< K, V > m )
```

Figura 20.21 | Métodos empacotadores não modificáveis.



COMO PROGRAMAR

8^a edição



Observação de engenharia de software 20.5

Você pode utilizar um empacotador não modificável para criar uma coleção que oferece acesso de leitura às outras pessoas enquanto permite o acesso de leitura e gravação para você. Você faz isso simplesmente dando às outras pessoas uma referência ao empacotador não modificável ao mesmo tempo em que retém para você uma referência à coleção original.



COMO PROGRAMAR

8^a edição

20.15 Implementações abstratas

- ▶ A estrutura de coleções fornece várias implementações abstratas de interfaces **Collection** a partir das quais você pode implementar rapidamente aplicativos personalizados completos.
- ▶ Elas incluem:
 - uma implementação simples de **Collection** chamada **AbstractCollection**.
 - uma implementação de **List** que permite acesso aleatório a seus elementos chamada **AbstractList**.
 - uma implementação de **Map** chamada **AbstractMap**.
 - uma implementação de **List** que permite acesso sequencial a seus elementos chamada **AbstractSequentialList**.
 - uma implementação de **Set** chamada **AbstractSet**.
 - uma implementação de **Queue** chamada **AbstractQueue**.