

19

Coleções



OBJETIVOS

- Neste capítulo, você aprenderá:
- O que são coleções.
- Como utilizar a classe `Arrays` para manipulações de array.
- Como utilizar implementações de estrutura de coleções (estrutura de dados pré-empacotada).
- Como utilizar algoritmos de estrutura de coleções para manipular (como `search`, `sort` e `fill`) coleções.
- Como utilizar as interfaces de estrutura de coleções para programar com coleções polimorficamente.
- Como utilizar iteradores para ‘percorrer’ uma coleção.
- Como utilizar tabelas de hash persistentes manipuladas com objetos da classe `Properties`.
- Como utilizar empacotadores de sincronização e de modificabilidade.



- 19.1** **Introdução**
- 19.2** **Visão geral das coleções**
- 19.3** **Classe Arrays**
- 19.4** **Interface Collection e classe Collections**
- 19.5** **Listas**
 - 19.5.1** **ArrayList e Iterator**
 - 19.5.2** **LinkedList**
 - 19.5.3** **Vector**
- 19.6** **Algoritmos de coleções**
 - 19.6.1** **Algoritmo sort**
 - 19.6.2** **Algoritmo shuffle**
 - 19.6.3** **Os algoritmos reverse, fill, copy, max e min**
 - 19.6.4** **Algoritmo binarySearch**
 - 19.6.5** **Algoritmos addAll, frequency e disjoint**

- 19.7 Classe Stack do pacote `java.util`**
- 19.8 Classe `PriorityQueue` e Interface `Queue`**
- 19.9 Conjuntos**
- 19.10 Mapas**
- 19.11 Classe `Properties`**
- 19.12 Coleções sincronizadas**
- 19.13 Coleções não-modificáveis**
- 19.14 Implementações abstratas**
- 19.15 Conclusão**

19.1 Introdução

- **Java Collections Framework:**
 - **Contêm estruturas de dados, interfaces e algoritmos pré-empacotados.**
 - **Utilizam genéricos.**
 - **Utilizam as estruturas de dados existentes.**
 - **Exemplo da reutilização de código.**
 - **Fornece componentes reutilizáveis.**



19.2 Visão geral das coleções

- **Coleção:**

- Estrutura de dados (objeto) que pode conter referências a outros objetos.

- **Estrutura das coleções:**

- As interfaces declaram as operações para os vários tipos de coleção.
- Fornece implementações de alta qualidade e de alto desempenho das estruturas de dados comuns.
- Permite a reutilização de software.
- Aprimorado com as capacidades genéricas no J2SE 5.0.
 - Verificação de tipos em tempo de compilação.



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 19.1 | Algumas interfaces de estrutura de coleção.



19.3 Classe Arrays

- **Classe Arrays:**
 - Fornece métodos **static** para manipulação de arrays.
 - Fornece métodos ‘de alto nível’:
 - Método **binarySearch** para pesquisar arrays classificados.
 - Método **equals** para comparar arrays.
 - Método **fill** para colocar valores em arrays.
 - Método **sort** para classificar arrays.



Resumo

UsingArrays.java

(1 de 4)

Linha 17

Linha 18

```

1 // Fig. 19.2: UsingArrays.java
2 // Utilizando arrays de Java.
3 import java.util.Arrays;
4
5 public class UsingArrays
6 {
7     private int intArray[] = { 1, 2, 3, 4, 5, 6 };
8     private double doubleArray[] = { 8.4, 9.3, 0.2, 7.9, 3.4 };
9     private int filledIntArray[], intArrayCopy[];
10
11 // construtor inicializa arrays
12 public UsingArrays()
13 {
14     filledIntArray = new int[ 10 ]; // cria array int com 10 elementos
15     intArrayCopy = new int[ intArray.length ];
16
17     Arrays.fill( filledIntArray, 7 ); // preenche com 7s
18     Arrays.sort( doubleArray ); // classifica do menor para o maior
19
20 // copia array intArray no array intArrayCopy
21     System.arraycopy( intArray, 0, intArrayCopy,
22         0, intArray.length );
23 } // fim do construtor UsingArrays
24

```

Utiliza o método estático `fill` da classe `Arrays` para preencher o array com 7s

Utiliza o método estático `sort` da classe `Arrays` para classificar elementos do array em ordem crescente

Utiliza o método `static arraycopy` da classe `System` a fim de copiar o array `intArray` para o array `intArrayCopy`



```
25 // gera saída de valores em cada array
26 public void printArrays()
27 {
28     System.out.print( "doubleArray: " );
29     for ( double doubleValue : doubleArray )
30         System.out.printf( "%.1f ", doubleValue );
31
32     System.out.print( "\nintArray: " );
33     for ( int intValue : intArray )
34         System.out.printf( "%d ", intValue );
35
36     System.out.print( "\nfilledIntArray: " );
37     for ( int intValue : filledIntArray )
38         System.out.printf( "%d ", intValue );
39
40     System.out.print( "\nintArrayCopy: " );
41     for ( int intValue : intArrayCopy )
42         System.out.printf( "%d ", intValue );
43
44     System.out.println( "\n" );
45 } // fim do método printArrays
46
47 // localiza valor no array intArray
48 public int searchForInt( int value )
49 {
50     return Arrays.binarySearch( intArray, value );
51 } // fim do método searchForInt
52
```

Resumo

UsingArrays.java

(2 de 4)

Linha 50

Utiliza o método `static binarySearch` da classe `Arrays` para realizar uma pesquisa binária no array



Resumo

Utiliza o método `static equals` da classe `Arrays` para determinar se valores dos dois arrays são equivalentes

a

Linhas 56 e 60

```
53 // compara conteúdo do array
54 public void printEquality()
55 {
56     boolean b = Arrays.equals( intArray, intArrayCopy );
57     System.out.printf( "intArray %s intArrayCopy\n",
58         ( b ? "==" : "!=" ) );
59
60     b = Arrays.equals( intArray, filledIntArray );
61     System.out.printf( "intArray %s filledIntArray\n",
62         ( b ? "==" : "!=" ) );
63 } // fim do método printEquality
64
65 public static void main( String args[] )
66 {
67     UsingArrays usingArrays = new UsingArrays();
68
69     usingArrays.printArrays();
70     usingArrays.printEquality();
71 }
```



```

72  int location = usingArrays.searchForInt( 5 );
73  if ( location >= 0 )
74      System.out.printf(
75          "Found 5 at element %d in intArray\n", location );
76  else
77      System.out.println( "5 not found in intArray" );
78
79  location = usingArrays.searchForInt( 8763 );
80  if ( location >= 0 )
81      System.out.printf(
82          "Found 8763 at element %d in intArray\n", location );
83  else
84      System.out.println( "8763 not found in intArray" );
85  } // fim do main
86 } // fim da classe UsingArrays

```

Resumo

UsingArrays.java

(4 de 4)

Saída do programa

```

doubleArray: 0.2 3.4 7.9 8.4 9.3
intArray: 1 2 3 4 5 6
filledIntArray: 7 7 7 7 7 7 7 7 7 7
intArrayCopy: 1 2 3 4 5 6

```

```

intArray == intArrayCopy
intArray != filledIntArray
Found 5 at element 4 in intArray
8763 not found in intArray

```



Erro comum de programação 19.1

Passar um array não-classificado para `binarySearch` é um erro de lógica — o valor retornado é indefinido.



19.4 Interface Collection e classe Collections

- **Interface Collection:**

- Interface-raiz na hierarquia de coleções.
- As interfaces **Set**, **Queue** e **List** estendem a interface **Collection**.
 - **Set** – a coleção não contém duplicatas.
 - **Queue** – a coleção representa uma linha de espera.
 - **List** – a coleção ordenada pode conter elementos duplicados.
- Contém operações de *grande volume*.
 - Adicionando, limpando, comparando e retendo objetos.
- Fornece o método para retornar um objeto **Iterator**.
 - Investiga a coleção e remove elementos da coleção.



Observação de engenharia de software 19.1

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



Observação de engenharia de software 19.2

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.



19.4 Interface `Collection` e classe `Collections` (*Continuação*)

- **Classe `Collections`:**
 - Fornece métodos `static` que manipulam coleções.
 - Implementa algoritmos para pesquisar, classificar e assim por diante.
 - As coleções podem ser manipuladas polimorficamente.
- **Coleção sincronizada.**
- **Coleção não-modificável.**



19.5 Listas

- **List:**
 - **Collection** ordenada que pode conter elementos duplicados.
 - Às vezes chamada *seqüência*.
 - Implementada via a interface **List**:
 - **ArrayList**
 - **LinkedList**
 - **Vector**

Dica de desempenho 19.1

ArrayLists comportam-se como **Vectors** sem sincronização e, portanto, executam mais rapidamente que **Vectors** porque **ArrayLists** não têm o overhead da sincronização de threads.



Observação de engenharia de software 19.3

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

19.5.1 ArrayList e Iterator

- **Exemplo de ArrayList:**
 - Demonstra as capacidades da interface **Collection**.
 - Posiciona dois arrays de **String** em **ArrayLists**.
 - Utiliza **Iterator** para remover elementos na **ArrayList**.



```
1 // Fig. 19.3: CollectionTest.java
2 // Usando a interface Collection.
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     private static final String[] colors =
11         { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
12     private static final String[] removeColors =
13         { "RED", "WHITE", "BLUE" };
14
15     // cria ArrayList, adiciona Colors a ela e a manipula
16     public CollectionTest()
17     {
18         List< String > list = new ArrayList< String >();
19         List< String > removeList = new ArrayList< String >();
20     }
```

Resumo

CollectionTest
.java

(1 de 3)

Linhas 18-19

Cria objetos `ArrayList` e atribui suas referências à variável `list` e a `removeList`, respectivamente



Resumo

Utiliza o método `List add` para adicionar objetos a `list` e a `removeList`, respectivamente

.java

Utiliza o método `List size` para obter o número de elementos em `ArrayList`

As linhas 23 e 27

Utiliza o método `List get` para recuperar valores de elementos individuais

Linha 33

Linha 36

O método `removeColors` recebe duas `Collections` como argumentos; a linha 36 passa duas `Lists`, que estende `Collection`, para esse método

```

21 // adiciona elementos no array colors a listar
22 for ( String color : colors )
23     list.add( color );
24
25 // adiciona elementos em removeColors para removeList
26 for ( String color : removeColors )
27     removeList.add( color );
28
29 System.out.println( "ArrayList: " );
30
31 // gera saída do conteúdo da lista
32 for ( int count = 0; count < list.size(); count++ )
33     System.out.printf( "%s ", list.get( count ) );
34
35 // remove cores contidas em removeList
36 removeColors( list, removeList );
37
38 System.out.println( "\n\nArrayList after calling removeColors: " );
39
40 // gera saída do conteúdo da lista
41 for ( String color : list )
42     System.out.printf( "%s ", color );
43 } // fim do construtor CollectionTest
44

```



Resumo

```

45 // remove cores especificadas em collection2 de collection1
46 private void removeColors(
47     Collection< String > collection1, Collection< String > col
48 {
49     // obtém iterador
50     Iterator< String > iterator = collection1.iterator();
51
52     // faz loop enquanto coleção tem itens
53     while ( iterator.hasNext() )
54
55         if ( collection2.contains( iterator.next() ) )
56             iterator.remove(); // remove color atual
57 } // fim do método removeColors
58
59 public static void main( String args[] )
60 {
61     new CollectionTest();
62 } // fim do main
63 } // fim da classe CollectionTest

```

O método `removeColors` permite que quaisquer `Collections` que contêm strings sejam passadas como argumentos para esse método

Obtém o iterador `Collection`

O método `Iterator hasNext` determina se `Iterator` contém mais elementos

O método `Iterator next` retorna uma referência ao próximo elemento

O método `Collection contains` determina se `collection2` contém o elemento retornado por `next`

Utiliza o método `Iterator remove` para remover `String` de `Iterator`

ArrayList:
MAGENTA RED WHITE BLUE CYAN

ArrayList after calling `removeColors`:
MAGENTA CYAN



Erro comum de programação 19.2

Se uma coleção for modificada por um de seus métodos depois de um iterador ser criado para essa coleção, o iterador se torna imediatamente inválido — qualquer operação realizada com o iterador depois desse ponto lança

ConcurrentModificationExceptions. Por essa razão, diz-se que os iteradores ‘falham rápido’.



19.5.2 `LinkedList`

- **Exemplo de `LinkedList`:**
 - Adiciona elementos de uma `List` a outra.
 - Converte `Strings` em letras maiúsculas.
 - Exclui um intervalo de elementos.



Resumo

ListTest.java

(1 de 4)

Linhas 17-18

Linha 22

```
1 // Fig. 19.4: ListTest.java
2 // Utilizando LinkLists.
3 import java.util.List;
4 import java.util.LinkedList;
5 import java.util.ListIterator;
6
7 public class ListTest
8 {
9     private static final String colors[] = { "black", "yellow",
10        "green", "blue", "violet", "silver" };
11     private static final String colors2[] = { "gold", "white",
12        "brown", "blue", "gray", "silver" };
13
14     // configura e manipula objetos LinkedList
15     public ListTest()
16     {
17         List< String > list1 = new LinkedList< String >();
18         List< String > list2 = new LinkedList< String >();
19
20         // adiciona elementos ao link da lista
21         for ( String color : colors )
22             list1.add( color );
23     }
```

Cria dois objetos
LinkedList

Utiliza o método List add para acrescentar
elementos do array colors ao final da list1



```
// add elements to list link2
```

```
for ( String color : colors2 )
```

```
list2.add( color );
```

Utiliza o método `List add` para acrescentar elementos no array `colors2` ao final da `list2`

```
list1.addAll( list2 ); // concatenate lists
```

```
list2 = null; // release resources
```

```
printList( list1 ); // print list1 elements
```

ListTest.java

Utiliza o método `List addAll` para acrescentar todos os elementos de `list2` ao final de `list1`

```
convertToUpperStrings( list1 ); // converte
```

```
printList( list1 ); // print list1 elements
```

```
System.out.print( "\nDeleting elements 4 to 6..." );
```

```
removeItems( list1, 4, 7 ); // remove items 4-7 from list
```

```
printList( list1 ); // print list1 elements
```

```
printReversedList( list1 ); // print list in reverse order
```

```
} // end ListTest constructor
```

Linha 20

Linha 28

Linha 42

```
// output List contents
```

```
public void printList( List< String > list )
```

```
{
```

```
System.out.println( "\nlist: " );
```

```
for ( String color : list )
```

```
System.out.printf( "%s ", color );
```

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

```
} // end method printList
```

O método `printList` permite que quaisquer `Lists` que contêm strings sejam passadas como argumentos para esse método



Resumo

```

52 // localiza objetos String e converte em letras maiúsculas
53 private void convertToUpperStrings( List< String > list )
54 {
55     ListIterator< String > iterator = list.listIterator();
56
57     while ( iterator.hasNext() )
58     {
59         String color = iterator.next(); // obtém item
60         iterator.set( color.toUpperCase() );
61     } // fim do while
62 } // fim do método convertToUpperStrings
63
64 // obtém sublista e utiliza método clear() para remover itens
65 private void removeItems( List< String > list )
66 {
67     list.subList( start, end ).clear(); // remove itens
68 } // fim do método removeItems
69
70 // imprime lista invertida
71 private void printReversedList( List< String > list )
72 {
73     ListIterator< String > iterator = list.listIterator( list.size() );
74 }

```

O método

Invoca o método List

listIterator para obter um iterador
bidirecional para a List

Invoca o método ListIterator

Invoca o método next de

Invoca o método ListIterator set para substituir a
String atual que o iterador referencia pela String
retornada pelo método toUpperCase

O método removeItems permite que quaisquer
strings sejam passadas como
argumentos para o método

Invoca o método List subList
para obter uma parte da List

Linha 65

O método printReversedList
permite que quaisquer Lists que
contêm strings sejam passadas como
argumentos para o método

Invoca o método List
listIterator com um argumento
que especifica a posição inicial para
obter um iterador bidirecional para a lista

```

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
82  public static void main( String args[] )
83  {
84      new ListTest();
85  } // fim do main
86 } // fim da classe ListTest

```

A condição **while** (linha 78) chama o método **hasPrevious** para determinar se há mais elementos ao percorrer a lista de trás para frente.

ListTest.java

Invoca o método **ListIterator** **previous** para obter o elemento anterior na lista

Linha 79

```

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

```

Saída do programa



19.5.2 `LinkedList` (Continuação)

- Método `static asList` da classe `Arrays`:
 - Visualiza um array como uma coleção `List`.
 - Permite que o programador manipule o array como se ele fosse uma lista.
 - Qualquer modificação feita por meio da visão `List` altera o array.
 - Qualquer modificação feita no array altera a visão `List`.
 - A única operação permitida na visão retornada por `asList` é *set*.



Resumo

UsingToArray.java

```

1 // Fig. 19.5: UsingToArray.java
2 // Utilizando o método toArray.
3 import java.util.LinkedList;
4 import java.util.Arrays;
5
6 public class UsingToArray
7 {
8     // construtor cria LinkedList, adiciona elementos e converte
9     public UsingToArray()
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
17        links.add( "pink" ); // adiciona ao final
18        links.add( 3, "green" ); // adiciona no terceiro
19        links.addFirst( "cyan" ); // adiciona como primeiro item
20    }

```

Chama o método `asList` para criar uma visão `List` do array `colors`, que é então utilizado para criar uma `LinkedList`

Linha 16

Chama o método `LinkedList` `addLast` para adicionar 'red' ao final de `links`

Chama o método `LinkedList` `add` para adicionar 'pink' como o último elemento e 'green' como o elemento no índice 3

Chama o método `LinkedList` `addFirst` para adicionar 'cyan' como o novo primeiro item na `LinkedList`



Resumo

Utiliza o método `List toArray` para obter uma representação de array da `LinkedList`

(2 de 2)

Linha 22

Saída do programa

```
21 // obtém elementos LinkedList como um array
22 colors = links.toArray( new String[ links.size() ] );
23
24 System.out.println( "colors: " );
25
26 for ( String color : colors )
27     System.out.println( color );
28 } // fim do construtor UsingToArray
29
30 public static void main( String args[] )
31 {
32     new UsingToArray();
33 } // fim do main
34 } // fim da classe UsingToArray
```

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



Erro comum de programação 19.3

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 sobrescritos pelos elementos da lista. Os elementos do array que não são sobrescritos retêm seus valores.



19.5.3 Vector

- **Classe Vector:**

- Fornece capacidades de estruturas de dados, como de array, que podem se redimensionar dinamicamente.
- Contém uma *capacidade*.
- Aumenta por *incremento de capacidade* se requerer espaço adicional.



Dica de desempenho 19.2

Inserir um elemento em um `Vector` cujo tamanho atual é menor que sua capacidade é uma operação relativamente rápida.



Dica de desempenho 19.3

Inserir um elemento em um Vector que precisa aumentar a fim de acomodar o novo elemento é uma operação relativamente lenta.



Dica de desempenho 19.4

O incremento de capacidade padrão dobra o tamanho do Vector. Isso pode parecer um desperdício de memória, mas é realmente uma maneira eficiente de muitos Vectors aumentarem de modo rápido para ter ‘aproximadamente o tamanho certo’. Essa operação é muito mais eficiente que ter toda vez de aumentar o Vector apenas pela quantidade de espaço que ele aceita para armazenar um único elemento. A desvantagem é que o Vector poderia ocupar mais espaço que o necessário. Esse é um exemplo clássico da relação de troca espaço-tempo.



Dica de desempenho 19.5

Se a memória for um recurso escasso, utilize o método `Vector trimToSize` para aparar a capacidade de um `Vector` ao tamanho exato do `Vector`. Essa operação otimiza a utilização de memória de um `Vector`. Entretanto, adicionar outro elemento ao `Vector` forçará o `Vector` a crescer dinamicamente (de novo, uma operação relativamente lenta) — a operação de aparar não deixa nenhum espaço para o crescimento.



Resumo

VectorTest.java

(1 de 4)

```
1 // Fig. 19.6: VectorTest.java
2 // Utilizando a classe Vector.
3 import java.util.Vector;
4 import java.util.NoSuchElementException;
5
6 public class VectorTest
7 {
8     private static final String colors[] = { "red", "white", "blue" };
9
10    public VectorTest()
11    {
12        Vector< String > vector = new Vector< String >();
13        printVector( vector ); // imprime vetor
14
15        // adiciona elementos ao vetor
16        for ( String color : colors )
17            vector.add( color );
18
19        printVector( vector ); // imprime
20    }
```

Cria **Vector** do tipo de **String** com a capacidade inicial de 10 elementos e incremento de capacidade de zero

Chame o método **Vector** **add** para adicionar objetos (**Strings** nesse exemplo) ao final de **Vector**



Resumo

```

21 // gera saída dos primeiros e últimos elementos
22 try
23 {
24     System.out.printf( "First element: %s\n", vector.firstElement());
25     System.out.printf( "Last element: %s\n", vector.lastElement() );
26 } // fim do try
27 // captura exceção se vetor estiver vazio
28 catch ( NoSuchElementException exce
29 {
30     exception.printStackTrace()
31 } // fim do catch
32
33 // o vetor contém "red"?
34 if ( vector.contains( "red" ) )
35     System.out.printf( "\n\"red\" found at
36         vector.indexOf( "red" ) );
37 else
38     System.out.println( "\n\"red\" not found" );
39
40 vector.remove( "red" ); // remove a string "red"
41 System.out.println( "\"red\" has been removed" );
42 printVector( vector ); // imprime vetor
43

```

Chama o método de **Vector** **firstElement** para retornar uma referência ao primeiro elemento no **Vector**.

Chama o método de **Vector** **lastElement** para retornar uma referência ao último elemento no **Vector**.

O método **Vector** **contains** retorna um **boolean** que indica se **Vector** contém um **Object** específico

O método **Vector** **remove** remove a primeira ocorrência do seu argumento **Object** de **Vector**

Linha 40

O método **Vector** **indexOf** retorna um índice da primeira localização em **Vector** que contém o argumento



Resumo

VectorTest.java

(3 de 4)

```

44 // o vetor contém "red" depois da operação remove?
45 if ( vector.contains( "red" ) )
46     System.out.printf(
47         "\"red\" found at index %d\n", vector.indexOf( "red" ) );
48 else
49     System.out.println( "\"red\" not found" );
50
51 // imprime o tamanho e capacidade de vetor
52 System.out.printf( "\nsize: %d\nCapacity: %d\n", vector.size(),
53     vector.capacity() );
54 } // fim do construtor Vector
55
56 private void printVector( Vector< String > vectorToOutput )
57 {
58     if ( vectorToOutput.isEmpty() )
59         System.out.print( "vector is empty" );
60     else // itera pelos elementos
61     {
62         System.out.print( "vector contains: " );
63
64         // gera saída dos elementos
65         for ( String element : vectorToOutput )
66             System.out.printf( "%s ", element );
67     } // fim de else
68

```

Os métodos **Vector** **size** e **capacity** retornam o número de elementos em **Vector** e na capacidade de **Vector** respectivamente

O método **printVector** permite que quaisquer **Vectors** que contêm strings sejam passados como argumentos para esse método

O método **Vector** **isEmpty** retorna **true** se não houver nenhum elemento no **Vector**



```
69     System.out.println( "\n" );  
70 } // fim do método printVector  
71  
72 public static void main( String args[] )  
73 {  
74     new VectorTest(); // cria o objeto e chama seu construtor  
75 } // fim do main  
76 } // fim da classe VectorTest
```

Resumo

VectorTest.java

(4 de 4)

Saída do programa

```
vector is empty  
vector contains: red white blue  
First element: red  
Last element: blue  
"red" found at index 0  
"red" has been removed  
vector contains: white blue  
"red" not found  
Size: 2  
Capacity: 10
```



Erro comum de programação 19.4

Sem sobrescrever o método `equals`, o programa realiza comparações utilizando operador `==` para determinar se duas referências referenciam o mesmo objeto na memória.



Dica de desempenho 19.6

Os métodos `Vector` `contains` e `indexOf` realizam pesquisas lineares no conteúdo de um `Vector`. Essas pesquisas são ineficientes para grandes `Vectors`. Se um programa costuma pesquisar elementos em uma coleção, considere utilizar uma das implementações `Map` da Java Collection API (Seção 19.10), que fornecem capacidades de pesquisa em alta velocidade.



19.6 Algoritmos de coleções

- O framework de `CoLlections` fornece um conjunto de algoritmos:
 - Implementados como métodos `static`.
 - Algoritmos de `LiSt`:
 - `sort`
 - `binarySearch`
 - `reverse`
 - `shuffle`
 - `fill`
 - `copy`



19.6 Algoritmos de coleções (Cont.)

- **Algoritmos de Collection:**
 - min
 - max
 - addAll
 - frequency
 - disjoint



Algoritmo	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 coleção.
frequency	Calcula quantos elementos na coleção são iguais ao elemento especificado.
disjoint	Determina se duas coleções não têm nenhum elemento em comum.

Figura 19.7 | Algoritmos da classe Collections.



Observação de engenharia de software 19.4

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

19.6.1 Algoritmo sort

- **Sort:**
 - **Classifica elementos de List:**
 - A ordem é determinada pela ordem natural do tipo dos elementos.
 - Elementos de List devem implementar a interface Comparable.
 - Ou passar um Comparator para o método sort.
- **Classificando em ordem crescente:**
 - Método Collections sort.
- **Classificando em ordem decrescente:**
 - Método estático reverseOrder de Collections.
- **Classificando com um Comparator:**
 - Cria uma classe Comparator personalizada.



Resumo

Sort1.java

(1 de 2)

Linha 15

```
1 // Fig. 19.8: Sort1.java
2 // Utilizando o algoritmo sort.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 public class Sort1
8 {
9     private static final String suits[] =
10         { "Hearts", "Diamonds", "Clubs", "Spades" };
11
12     // exibe elementos do array
13     public void printElements()
14     {
15         List< String > list = Arrays.asList( suits ); // cria a
16
```

Cria List de Strings



Resumo

Chamada implícita ao método `toString` de `list` para gerar saída do conteúdo da lista

Utiliza o algoritmo `sort` para ordenar os elementos da lista em ordem crescente

Saída do programa

```

17 // gera saída da lista
18 System.out.printf( "Unsorted array elements:\n%s\n", list );
19
20 Collections.sort( list ); // classifica ArrayList
21
22 // gera saída da lista
23 System.out.printf( "Sorted array elements:\n%s\n", list );
24 } // fim do método printElements
25
26 public static void main( String args[] )
27 {
28     Sort1 sort1 = new Sort1();
29     sort1.printElements();
30 } // fim do main
31 } // fim da classe Sort1
  
```

```

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



Resumo

Sort2.java

(1 de 2)

```
1 // Fig. 19.9: Sort2.java
2 // Utilizando um objeto Comparador com o algoritmo sort.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 public class Sort2
8 {
9     private static final String suits[] =
10         { "Hearts", "Diamonds", "Clubs", "Spades" };
11
12     // gera saída de elementos List
13     public void printElements()
14     {
15         List list = Arrays.asList( suits ); // cria List
16     }
```



Resumo

Sort2.java

```
17 // gera saída dos elementos de List
18 System.out.printf( "Unsorted array elements:\n%s\n", list );
19
20 // classifica em ordem decrescente utilizando um comparador
21 Collections.sort( list, Collections.reverseOrder() );
22
23 // gera saída dos elementos de List
24 System.out.printf( "Sorted list elements:\n%s\n", list );
25 } // fim do método printElements
26
27 public static void main( String args[] )
28 {
29     Sort2 sort2 = new Sort2();
30     sort2.printElements();
31 } // fim de main
32 } // fim da classe Sort2
```

O método `reverseOrder` da classe `Collections` retorna um objeto `Comparator` que representa a ordem inversa da coleção

O método `sort` da classe `Collections` pode utilizar um objeto `Comparator` para classificar uma `List`

```
Unsorted array elements:
[Hearts, Diamonds, Clubs, Spades]
Sorted list elements:
[Spades, Hearts, Diamonds, Clubs]
```



Resumo

```

1 // Fig. 19.10: TimeComparator.java
2 // Classe Comparator personalizada que compara dois objetos Time2.
3 import java.util.Comparator;
4
5 public class TimeComparator implements Comparator< Time2 >
6 {
7     public int compare( Time2 tim1, Time2 time2 )
8     {
9         int hourCompare = tim1.getHour() - time2.getHour();
10
11         // testa a primeira hora
12         if ( hourCompare != 0 )
13             return hourCompare;
14
15         int minuteCompare =
16             tim1.getMinute() - time2.getMinute(); // compara minuto
17
18         // então testa o minuto
19         if ( minuteCompare != 0 )
20             return minuteCompare;
21
22         int secondCompare =
23             tim1.getSecond() - time2.getSecond(); // compara segundo
24
25         return secondCompare; // retorna o resultado da comparação de segundos
26     } // fim do método compare
27 } // fim da classe TimeComparator

```

O comparador personalizado **TimeComparator** implementa a interface **Comparator** e compara o objeto **Time2**

Implementa o método **compare** para determinar a ordem dos dois objetos **Time2**



```
1 // Fig. 19.11: Sort3.java
2 // Classifica uma lista usando a classe Comparator TimeComparator personalizada.
3 import java.util.List;
4 import java.util.ArrayList;
5 import java.util.Collections;
6
7 public class Sort3
8 {
9     public void printElements()
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    }
```

Resumo

Sort3.java

(1 de 2)



Resumo

Sort3.java

Classifica na ordem utilizando um comparador personalizado
TimeComparator

Saída do programa

```

19 // gera elementos de List
20 System.out.printf( "Unsorted array elements:\n%s\n", list );
21
22 // classifica em ordem utilizando um comparador
23 Collections.sort( list, new TimeComparator() );
24
25 // gera saída de elementos List
26 System.out.printf( "Sorted list elements:\n%s\n", list );
27 } // fim do método printElements
28
29 public static void main( String args[] )
30 {
31     Sort3 sort3 = new Sort3();
32     sort3.printElements();
33 } // fim de main
34 } // fim da classe Sort3
  
```

```

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]
  
```



19.6.2 Algoritmo shuffle

- **shuffle:**
 - Ordena aleatoriamente elementos de **List**.



Resumo

DeckOfCards.java

(1 de 4)

```
1 // Fig. 19.12: DeckOfCards.java
2 // Utilizando o algoritmo shuffle.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 // classe para representar uma Card 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; // naipes 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 naipes da carta
22     } // construtor Card de dois argumentos
23
24     // retorna a face da carta
25     public Face getFace()
26     {
27         return face;
28     } // fim do método getFace
29 }
```



Resumo

DeckOfCards.java

(2 de 4)

```
30 // retorna naipe de card
31 public Suit getSuit()
32 {
33     return suit;
34 } // fim do método getSuit
35
36 // retorna representação de 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
43 // declaração da classe DeckOfCards
44 public class DeckOfCards
45 {
46     private List< Card > list; // declara List que armazenará Cards
47
48     // configura baralho de Cards e embaralha
49     public DeckOfCards()
50     {
51         Card[] deck = new Card[ 52 ];
52         int count = 0; // número de cartas
53
```



Resumo

s.java

```

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     } // fim do for
62 } // fim do for

63
64 list = Arrays.asList( deck ); // obtém List
65 Collections.shuffle( list ); // embaralha as
66 } // fim do construtor DeckOfCards

67
68 // gera saída do 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( "%-20s%s", list.get( i ),
74                             ( ( i + 1 ) % 2 == 0 ) ? "\n" : "\t" );
75 } // fim do método printCards

76
77 public static void main( String args[] )
78 {
79     DeckOfCards cards = new DeckOfCards();
80     cards.printCards();
81 } // fim do main
82 } // fim da classe DeckOfCards

```

Utiliza o tipo **enum Suit** fora da classe **Card**, qualifica o nome de tipo de **enum (Suit)** com o nome da classe **Card** e um separador ponto (.)

qualifica o nome de tipo de **enum (Face)** com o nome da classe **Card** e um separador ponto

Invoca o método **static asList** da classe **Arrays** para obter uma visão **List** do array **deck**

Utiliza o método **shuffle** da classe **Collections** para embaralhar **List**



Resumo

DeckOfCards.java

(4 de 4)

Saída do programa

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



19.6.3 Os algoritmos reverse, fill, copy, max e min

- **reverse**

- Inverte a ordem dos elementos de `List`.

- **fill**

- Preenche os elementos de `List` com valores.

- **copy**

- Cria uma cópia de uma `List`.

- **max**

- Retorna o maior elemento em `List`.

- **min**

- Retorna o menor elemento em `List`.



Resumo

Algorithms1.java

(1 de 3)

Linha 24

```
1 // Fig. 19.13: Algorithms1.java
2 // Utilizando algoritmos reverse, fill, copy, min e max.
3 import java.util.List;
4 import java.util.Arrays;
5 import java.util.Collections;
6
7 public class Algorithms1
8 {
9     private Character[] letters = { 'P', 'C', 'M' };
10    private Character[] lettersCopy;
11    private List< Character > list;
12    private List< Character > copyList;
13
14    // cria uma List e a manipula com métodos de Collections
15    public Algorithms1()
16    {
17        list = Arrays.asList( letters ); // obtém List
18        lettersCopy = new Character[ 3 ];
19        copyList = Arrays.asList( lettersCopy ); // visualização de lista de lettersCopy
20
21        System.out.println( "Initial list: " );
22        output( list );
23
24        Collections.reverse( list ); // inverte a ordem
25        System.out.println( "\nAfter calling reverse: " );
26        output( list );
27    }
```

Utiliza o método `reverse` da classe `Collections` para obter `List` na ordem inversa




```
28 Collections.copy( copyList, list ); // copia List
29 System.out.println( "\nAfter copying: " );
30 output( copyList );
31
32 Collections.fill( list, 'R' ); // preenche a lista com
33 System.out.println( "\nAfter calling fill: " );
34 output( list );
35 } // fim do construtor Algorithms1
36
37 // gera saída de informações de List
38 private 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 );
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
```

Utiliza o método **copy** da classe **Collections** para obter uma cópia da **List**

Algorithms1.java

Utiliza o método **fill** da classe **Collections** para preencher **List** com a letra 'R'

Linha 32

Linha 45

Obtém o valor máximo na **List**

Obtém o valor mínimo na **List**



```
49 public static void main( String args[] )
50 {
51     new Algorithms1();
52 } // fim do main
53 } // fim da classe Algorithms1
```

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

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

After copying:
The list is: M C P
Max: P   Min: C

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

Resumo

Algorithms1.java

(3 de 3)

Saída do programa



19.6.4 Algoritmo `binarySearch`

- **`binarySearch`:**

- Localiza um objeto na `List`.
 - Retorna um índice de objetos na `List` se houver um objeto.
 - Retorna um valor negativo se `Object` não existir.
 - Calcula o ponto de inserção.
 - Torna negativo o sinal do ponto de inserção.
 - Subtrai 1 do ponto de inserção.



Resumo

BinarySearchTest
.java

(1 de 3)

Linha 18

```
1 // Fig. 19.14: BinarySearchTest.java
2 // Utilizando o algoritmo 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     private static final String colors[] = { "red", "white",
11         "blue", "black", "yellow", "purple", "tan", "pink" };
12     private List< String > list; // referência ArrayList
13
14     // cria, classifica e gera saída da lista
15     public BinarySearchTest()
16     {
17         list = new ArrayList< String >( Arrays.asList( colors ) );
18         Collections.sort( list ); // classifica a ArrayList
19         System.out.printf( "Sorted ArrayList: %s\n", list );
20     } // fim do construtor BinarySearchTest
21
```

Classifica List em ordem
crescente



Resumo

BinarySearchTest
.java

(2 de 3)

Linha 39

```
22 // pesquisa vários valores na lista
23 private void search()
24 {
25     printSearchResults( colors[ 3 ] ); // primeiro item
26     printSearchResults( colors[ 0 ] ); // item intermediário
27     printSearchResults( colors[ 7 ] ); // último item
28     printSearchResults( "aqua" ); // abaixo do mais baixo
29     printSearchResults( "gray" ); // não existe
30     printSearchResults( "teal" ); // não existe
31 } // fim do método search
32
33 // realiza pesquisas e exibe resultado da pesquisa
34 private void printSearchResults( String key )
35 {
36     int result = 0;
37
38     System.out.printf( "\nSearching for: %s\n", key );
39     result = Collections.binarySearch( list, key );
40
41     if ( result >= 0 )
42         System.out.printf( "Found at index %d\n", result );
43     else
44         System.out.printf( "Not Found (%d)\n", result );
45 } // fim do método printSearchResults
46
```

Utiliza o método
binarySearch da classe
collections para pesquisar
uma **key** especificada na **list**



```
47 public static void main( String args[] )
48 {
49     BinarySearchTest binarySearchTest = new BinarySearchTest();
50     binarySearchTest.search();
51 } // fim de main
52 } // fim da classe BinarySearchTest
```

Resumo

BinarySearchTest
.java

(3 de 3)

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)



19.6.5 Algoritmos `addAll`, `frequency` e `disjoint`

- **`addAll`**

- Insere todos os elementos de um array em uma coleção.

- **`frequency`**

- Calcula o número de vezes que um elemento específico aparece na coleção.

- **`Disjoint`**

- Determina se duas coleções não têm nenhum elemento em comum.



Resumo

Algorithms2.java

(1 de 3)

```
1 // Fig. 19.15: Algorithms2.java
2 // Utilizando os algoritmos addAll, frequency e disjoint.
3 import java.util.List;
4 import java.util.Vector;
5 import java.util.Arrays;
6 import java.util.Collections;
7
8 public class Algorithms2
9 {
10     private String[] colors = { "red", "white", "yellow", "blue" };
11     private List< String > list;
12     private Vector< String > vector = new Vector< String >();
13
14     // cria List e Vector
15     // e os manipula com os métodos de Collections
16     public Algorithms2()
17     {
18         // inicializa list e vector
19         list = Arrays.asList( colors );
20         vector.add( "black" );
21         vector.add( "red" );
22         vector.add( "green" );
23
24         System.out.println( "Before addAll, vector contains: " );
25     }
```



Resumo

Algorithms2.java

(2 de 3)

Linha 31

Linha 40

```
26 // exibe os elementos em vector
27 for ( String s : vector )
28     System.out.printf( "%s ", s );
29
30 // adiciona elementos de colors à list
31 Collections.addAll( vector, colors );
32
33 System.out.println( "\n\nAfter addAll, vector
34
35 // exibe elementos em vector
36 for ( String s : vector )
37     System.out.printf( "%s ", s );
38
39 // obtém frequência de "red"
40 int frequency = Collections.frequency( vector, "red" );
41 System.out.printf(
42     "\n\nFrequency of red in vector: %d\n", frequ
43
```

Invoca o método `addAll` para adicionar elementos no array `colors` a `vector`

Obtém a frequência da String 'red' em Collection `vector` utilizando o método `frequency`



(3 de 3)

Linha 45

Invoca o método `disjoint` para testar se `Collections` `list` e `vector` têm elementos em comum

```
44 // verifica se list e vector têm elementos em comum
45 boolean disjoint = Collections.disjoint( list, vector );
46
47 System.out.printf( "\nlist and vector %s elements
48     ( disjoint ? "do not have" : "have" ) );
49 } // fim do construtor Algorithms2
50
51 public static void main( String args[] )
52 {
53     new Algorithms2();
54 } // fim do main
55 } // fim da classe Algorithms2
```

Before addAll, vector contains:
black red green

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

Frequency of red in vector: 2

list and vector have elements in common



19.7 Classe Stack do pacote `java.util`

- **Stack:**

- **Implementa a estrutura de dados de pilha.**
- **Estende a classe `Vector`.**
- **Armazena referências a objetos.**



Resumo

Stacktest.java

(1 de 4)

Criam uma Stack vazia
do tipo Number

Linhas 19, 21, 23, 25

O método Stack push
adiciona objetos na parte
superior da Stack

```

1 // Fig. 19.16: StackTest.java
2 // Programa para testar java.util.Stack.
3 import java.util.Stack;
4 import java.util.EmptyStackException;
5
6 public class StackTest
7 {
8     public StackTest()
9     {
10         Stack< Number > stack = new Stack< Number >();
11
12         // cria números para armazenar na pilha
13         Long longNumber = 12L;
14         Integer intNumber = 34567;
15         Float floatNumber = 1.0F;
16         Double doubleNumber = 1234.5678;
17
18         // utiliza o método push
19         stack.push( longNumber ); // adiciona um long
20         printStack( stack );
21         stack.push( intNumber ); // adiciona um int
22         printStack( stack );
23         stack.push( floatNumber ); // adiciona um float
24         printStack( stack );
25         stack.push( doubleNumber ); // adiciona um double
26         printStack( stack );
27

```



Resumo

stacktest.java

(2 de 4)

Linha 36

Linha 49

```

28 // remove itens da pilha
29 try
30 {
31     Number removedObject = null;
32
33     // remove elementos da pilha
34     while ( true )
35     {
36         removedObject = stack.pop(); // utiliza o método pop
37         System.out.printf( "%s popped\n", removedObject );
38         printStack( stack );
39     } // fim do while
40 } // fim do try
41 catch ( EmptyStackException emptyStackException )
42 {
43     emptyStackException.printStackTrace();
44 } // fim do catch
45 } // fim do construtor StackTest
46
47 private void printStack( Stack< Number
48 {
49     if ( stack.isEmpty() )
50         System.out.print( "stack is empty\n\n" ); // a pilha está vazia
51     else // a pilha não está vazia
52     {
53         System.out.print( "stack contains: " );
54

```

O método Stack pop
remove elemento de parte
superior da Stack

O método Stack isEmpty
retorna true se Stack estiver
vazia



Resumo

Stacktest.java

(3 de 4)

```
55     // itera pelos elementos
56     for ( Number number : stack )
57         System.out.printf( "%s ", number );
58
59         System.out.print( "(top) \n\n" ); // indica a parte superior da pilha
60     } // fim de else
61 } // fim do método printStack
62
63 public static void main( String args[] )
64 {
65     new StackTest();
66 } // fim do main
67 } // fim da classe StackTest
```



Resumo

Stacktest.java

(4 de 4)

Saída do programa

```
stack contains: 12 (top)
stack contains: 12 34567 (top)
stack contains: 12 34567 1.0 (top)
stack contains: 12 34567 1.0 1234.5678 (top)
1234.5678 popped
stack contains: 12 34567 1.0 (top)
1.0 popped
stack contains: 12 34567 (top)
34567 popped
stack contains: 12 (top)
12 popped
stack is empty
java.util.EmptyStackException
    at java.util.Stack.peek(Unknown Source)
    at java.util.Stack.pop(Unknown Source)
    at StackTest.<init>(StackTest.java:36)
    at StackTest.main(StackTest.java:65)
```



Dica de prevenção de erro 19.1

Como Stack estende Vector, todos os métodos Vector públicos podem ser chamados nos 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 e remover elementos da Stack, respectivamente.



19.8 Classe `PriorityQueue` e Interface `Queue`

- **Interface `Queue`:**

- Nova interface de coleção introduzida no J2SE 5.0.
- Estende a interface `Collection`.
- Fornece operações adicionais para inserir, remover e inspecionar elementos em uma fila.

- **Classe `PriorityQueue`:**

- Implementa a interface `Queue`.
- Ordena os elementos de acordo com seu ordenamento natural.
 - Especificada pelo método `compareTo` dos elementos de `Comparable`.
 - Objeto `Comparator` fornecido pelo construtor.



Resumo

1 // Fig. 19.17: PriorityQueueTest.java

2 // Programa de teste da classe de biblioteca padrão 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 >();

11
12 // insere elementos à 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

19 // exibe elementos na fila

20 while (queue.size() > 0)

21 {

22 System.out.printf("%.1f ", queue.peek());

23 queue.poll(); // remove elemento superior

24 } // fim do while

25 } // fim do main

26 } // fim da classe PriorityQueueTest

Cria uma **PriorityQueue** que armazena **Doubles** com uma capacidade inicial de 11 elementos e ordena os elementos de acordo com o ordenamento natural do objeto

.java

Linha 10

Linhas 13-15

Linha 20

Linha 22

Utiliza o método **offer** para adicionar elementos à fila de prioridades

Utiliza o método **size** para determinar se a fila de prioridades está vazia

Linha 23

Fim do programa

Utiliza o método **peek** para recuperar o elemento com a prioridade mais alta na fila

Utiliza o método **poll** para remover o elemento com a prioridade mais alta da fila

Polling from queue: 3.2 5.4 9.8



19.9 Conjuntos

- **Set:**
 - **Collection** que contém elementos únicos.
 - **HashSet:**
 - Armazena os elementos na tabela de hash.
 - **TreeSet:**
 - Armazena os elementos na árvore.



Resumo

SetTest.java

(1 de 2)

Linha 18

```
1 // Fig. 19.18: SetTest.java
2 // Utilizando um HashSet para remover duplicatas.
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     private static final String colors[] = { "red", "white", "blue",
12         "green", "gray", "orange", "tan", "white", "cyan",
13         "peach", "gray", "orange" };
14
15     // cria ArrayList e gera sua saída
16     public SetTest()
17     {
18         List< String > list = Arrays.asList( colors );
19         System.out.printf( "ArrayList: %s\n", list );
20         printNonDuplicates( list );
21     } // fim do construtor SetTest
22 }
```

Cria uma `List` que contém objetos `String`



Resumo

SetTest.java

(2 de 2)

Linha 24

Linha 27

Saída do programa

```
23 // cria conjunto de array para eliminar duplicatas
24 private void printNonDuplicates( Collection< String > collection )
25 {
26     // cria um HashSet
27     Set< String > set = new HashSet< String >( collection );
28
29     System.out.println( "\nNonduplicates are: " );
30
31     for ( String s : set )
32         System.out.printf( "%s ", s );
33
34     System.out.println();
35 } // fim do método printNonDuplicates
36
37 public static void main( String args[] )
38 {
39     new SetTest();
40 } // fim do main
41 } // fim da classe SetTest
```

Constrói um HashSet a
partir do argumento de
collection

O método printNonDuplicates aceita uma
Collection do tipo String

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

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



Resumo

SortedSetTest
.java

(1 de 3)

Linhas 16-17

```
1 // Fig. 19.19: SortedSetTest.java
2 // Utilizando TreeSet e SortedSet.
3 import java.util.Arrays;
4 import java.util.SortedSet;
5 import java.util.TreeSet;
6
7 public class SortedSetTest
8 {
9     private static final String names[] = { "yellow", "green",
10         "black", "tan", "grey", "white", "orange", "red", "green" };
11
12     // cria um conjunto classificado com TreeSet, e depois o manipula
13     public SortedSetTest()
14     {
15         // cria o TreeSet
16         SortedSet< String > tree =
17             new TreeSet< String >( Arrays.asList( names ) );
18
19         System.out.println( "sorted set: " );
20         printSet( tree ); // gera saída do conteúdo da árvore
21     }
22 }
```

Cria um TreeSet
a partir do array de
names



```
// obtém headSet com base em "orange"
```

```
System.out.print( "\nheadSet (\"orange\"):" );
printSet( tree.headSet( "orange" ) );
```

Utiliza o método `TreeSet headSet` para obter um subconjunto de `TreeSet` menor que 'orange'

```
// obtém tailSet baseado em "orange"
```

```
System.out.print( "tailSet (\"orange\"):" );
printSet( tree.tailSet( "orange" ) );
```

Utiliza o método `TreeSet tailSet` para obter o subconjunto de `TreeSet` maior que 'orange'

```
// obtém primeiro e últimos elementos
```

```
System.out.printf( "first: %s\n", tree.first() );
```

```
System.out.printf( "last : %s\n", tree.last() );
```

```
} // fim do construtor SortedSetTest
```

Os métodos `first` e `last` obtêm o maior e o menor elemento de `TreeSet`, respectivamente

```
// gera saída do conteúdo
```

```
private void printSet( SortedSet< String > set )
```

```
{
```

```
    for ( String s : set )
```

```
        System.out.printf( "%s ", s );
```

Linha 28

Linha 31

Linha 32



```
41     System.out.println();
42 } // fim do método printSet
43
44 public static void main( String args[] )
45 {
46     new SortedSetTest();
47 } // fim do main
48 } // fim da classe SortedSetTest
```

```
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
```

Resumo

SortedSetTest
.java

(3 de 3)

Saída do programa



19.10 Mapas

- **Map:**

- Associa chaves a valores.
- Não pode conter chaves duplicadas.
 - Chamado *mapeamento um para um*.
- Classes de implementação:
 - **Hashtable, HashMap:**
 - Armazena elementos nas tabelas de hash.
 - **TreeMap:**
 - Armazena elementos nas árvores.
- Interface **SortedMap:**
 - Estende Map.
 - Mantém suas chaves na ordem classificada.



19.10 Mapas (Cont.)

- **Implementação de map com tabelas de hash:**
 - Tabelas de hash.
 - Estrutura de dados que utiliza *hashing*.
 - Algoritmo para determinar uma *chave* na tabela.
 - Chaves nas tabelas têm valores associados (dados).
 - Cada célula na tabela é um ‘bucket’ (recipiente) de hash.
 - Lista vinculada de todos os *pares chave/valor* que sofrem hash para essa célula.
 - Minimiza *colisões*.



Dica de desempenho 19.7

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 devido ao aumento das colisões de hashing. Diminuindo o fator de carga, melhoramos a velocidade do programa, devido à redução das colisões de hashing, mas fazemos uma pobre utilização da memória porque uma parte maior da tabela de hash permanece vazia.



```
1 // Fig. 19.20: WordTypeCount.java
2 // Programa conta o número de ocorrências de cada palavra em uma string
3 import java.util.StringTokenizer;
4 import java.util.Map;
5 import java.util.HashMap;
6 import java.util.Set;
7 import java.util.TreeSet;
8 import java.util.Scanner;
9
10 public class WordTypeCount
11 {
12     private Map< String, Integer > map;
13     private Scanner scanner;
14
15     public WordTypeCount()
16     {
17         map = new HashMap< String, Integer >(); // cria HashMap
18         scanner = new Scanner( System.in ); // cria scanner
19         createMap(); // cria mapa com base na entrada do usuário
20         displayMap(); // exibe conteúdo do mapa
21     } // fim do construtor WordTypeCount
22
```

Resumo

WordTypeCount
.java

(1 de 4)

Linha 17

Cria um **HashMap** vazio com uma capacidade padrão de 16 e um fator de carga padrão de 0,75. As chaves são do tipo **String** e os valores são do tipo **Integer**



// cria mapa a partir da entrada do usuário

private void createMap()

{

System.out.println("Enter a string: ", // solicita a entrada de usuário

String input = scanner.nextLine(); // cria um StringTokenizer baseado no S

// cria StringTokenizer

StringTokenizer tokenizer = new StringTokenizer(input);

// processar

while (tok

{

String word = tokenizer.nextToken().toLowerCase(); // obtém a palavra

// se o mapa contiver a palavra

if (map.containsKey(word)) // palavra está no mapa

{

int count = map.get(word); // obt

map.put(word, count + 1); // inc

} // fim do if

else

map.put(word, 1); // adiciona

} // fim do while

} // fim do método createMap

Cria um StringTokenizer para dividir o argumento de entrada do tipo string nas suas palavras componentes individuais

Utiliza nextToken do método StringTokenizer para obter o próximo token

O método Map.containsKey determina se a chave especificada como um argumento está na tabela de hash

Utiliza o método get para obter o valor associado da Incrementa o valor e utiliza o método put para substituir o valor associado da chave

Cria uma nova entrada no mapa, com a palavra como a chave e um objeto Integer contendo 1 como o valor

(2 de 4)

Linha 30

Linha 33

Linha 35

Linha 44



Chama o método `IsEmpty` para determinar se o `Map` está vazio

Linha 63

```
66 public static void main( String args[] )
67 {
68     new WordTypeCount();
69 } // fim do main
70 } // fim da classe wordTypeCount
```

Resumo

WordTypeCount
.java

(4 de 4)

Saída do programa

```
Enter a string:
To be or not to be: that is the question whether 'tis nobler to suffer
Map contains:
Key                value
'tis                1
be                  1
be:                 1
is                  1
nobler              1
not                 1
or                  1
question            1
suffer              1
that                1
the                 1
to                  3
whether               1

size:13
isEmpty:false
```



19.11 A classe Properties

- **Properties:**

- **Hashtable** persistente.
 - Pode ser gravada no fluxo de saída.
 - Pode ser lida no fluxo de entrada.
- Fornece os métodos **setProperty** e **getProperty**.
 - Armazena/obtem os pares chave/valor de **Strings**.

- **Preferences API:**

- Substitui **Properties**.
- Mecanismo mais robusto.



Resumo

PropertiesTest
.java

(1 de 5)

Linha 16

Linhas 19-20

```
1 // Fig. 19.21: PropertiesTest.java
2 // Demonstra a 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     private Properties table;
12
13     // obtém GUI para testar a tabela Properties
14     public PropertiesTest()
15     {
16         table = new Properties(); // cria a tabela Properties
17
18         // configura as propriedades
19         table.setProperty( "color", "blue" );
20         table.setProperty( "width", "200" );
21
22         System.out.println( "After setting properties" );
23         listProperties(); // exibe valores da propriedade
24
25         // substitui o valor da propriedade
26         table.setProperty( "color", "red" );
27     }
28 }
```

Cria Properties vazia

O método Properties
setProperty armazena o valor para a
chave especificada



Resumo

PropertiesTest
.java

(2 de 5)

Linha 33

```
28 System.out.println( "After replacing properties" );
29 listProperties(); // exibe os valores da propriedade
30
31 saveProperties(); // salva as propriedades
32
33 table.clear(); // tablea vazia
34
35 System.out.println( "After clearing
36 listProperties(); // exibe valores da propriedade
37
38 loadProperties(); // carrega propriedade
39
40 // obtém valor de cor da propriedade
41 Object value = table.getProperty( "color" );
42
43 // verifica se o valor está na tabela
44 if ( value != null )
45     System.out.printf( "Property color's value is %s\n", value );
46 else
47     System.out.println( "Property color is not in table" );
48 } // fim do construtor PropertiesTest
49
```

Utiliza o método **Properties**
clear para esvaziar a tabela de
hash

Chama o método **Properties**
getProperty para localizar o
valor associado com a chave
especificada.



Resumo

PropertiesTest
.java

(3 de 5)

Linha 57

```
50 // salva propriedades em um arquivo
51 public void saveProperties()
52 {
53     // salva conteúdo da tabela
54     try
55     {
56         FileOutputStream output = new FileOutputStream( "props.dat" );
57         table.store( output, "Sample Properties" ); // salva propriedades
58         output.close();
59         System.out.println( "After saving" );
60         listProperties();
61     } // fim do try
62     catch ( IOException ioException )
63     {
64         ioException.printStackTrace();
65     } // fim do catch
66 } // fim do método saveProperties
67
```

O método Properties
store salva o conteúdo de
Properties em
FileOutputStream



Resumo

PropertiesTest
.java

```

68 // carrega propriedades de um arquivo
69 public void loadProperties()
70 {
71     // carrega conteúdo da tabela
72     try
73     {
74         FileInputStream input = new FileInputStream( "props.dat" );
75         table.load( input ); // carrega propriedades
76         input.close();
77         System.out.println( "After loading properties" );
78         listProperties(); // exibe os valores da tabela
79     } // fim do try
80     catch ( IOException ioException )
81     {
82         ioException.printStackTrace();
83     } // fim do catch
84 } // fim do método loadProperties
85
86 // gera saída de valores de propriedade
87 public void listProperties()
88 {
89     Set< Object > keys = table.keySet(); // obtém os nomes das propriedades
90
91     // gera saída de pares nome/valor
92     for ( Object key : keys )
93     {
94         System.out.printf(
95             "%s\t%s\n", key, table.getProperty( ( String ) key ) );
96     } // fim de for
97

```

O método **Properties load** restaura o conteúdo de **Properties** a partir de **FileInputStream**

Linha 89

Linha 95

Utiliza o método **Properties keySet** para obter um **Set** dos nomes das propriedades

Obtém o valor de uma propriedade passando uma chave para o método **getProperty**



Resumo

PropertiesTest .java

(5 de 5)

Saída do programa

```
98      System.out.println();
99  } // fim do método listProperties
100
101  public static void main( String args[] )
102  {
103      new PropertiesTest();
104  } // fim do main
105} // fim da classe PropertiesTest
```

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



19.12 Coleções sincronizadas

- **Coleções predefinidas são não-sincronizadas.**
 - Acesso concorrente a uma `Collection` pode causar erros.
 - O Java fornece *empacotadores de sincronização* para evitar tais resultados indeterminados ou erros fatais.
 - Isso por meio do conjunto dos métodos `public static`.



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 19.22 | Métodos empacotadores de sincronização.



19.13 Coleções não-modificáveis

- **Empacotador não-modificável:**
 - Convertendo coleções em coleções não-modificáveis.
 - Lança uma `UnsupportedOperationException` caso sejam feitas tentativas para modificar a coleção.



Observação de engenharia de software 19.5

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 si mesmo. Você faz isso simplesmente dando às outras pessoas uma referência ao empacotador não-modificável ao mesmo tempo em que é retida uma referência à coleção original.

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 19.23 | Métodos empacotadores não-modificáveis.



19.14 Implementações abstratas

- **Implementações abstratas:**
 - Oferece uma implementação básica das interfaces de coleção.
 - Os programadores podem aprimorar as implementações personalizáveis.
 - `AbstractCollection`.
 - `AbstractList`.
 - `AbstractMap`.
 - `AbstractSequentialList`.
 - `AbstractSet`.
 - `AbstractQueue`.

