Java - Genéricos

BCC 221 - Programação Orientada a Objectos(POO)

Guillermo Cámara-Chávez

Departamento de Computação - UFOP

Coleções

- Uma coleção é um objeto que mantém referências a outros objetos
- As interfaces do *collections framework* declaram operações que podem ser realizadas genericamente em vários tipos de coleções
  - Pacote *java.util*
  - Várias implementações destas interfaces são fornecidas pelo *framework*
  - Podemos também criar nossas próprias implementações.

Coleções (cont.)

Interface	Descrição
Collection	A classe raiz na hierarquia de coleções, a partir da qual todas as outras são derivadas.
Set	Uma coleção que não contém repetições.
List	Uma coleção ordenada que pode conter repetições.
Map	Associa chaves a valores e não pode conter chaves duplicadas.
Queue	Coleção FIFO que modela uma fila, embora outras ordens possam ser especificada.

Classe *Arrays*

- A classe *Arrays* fornece métodos estáticos para manipular vetores
  - **sort**: ordena vetores (sobrecarregado com versões genéricas);
  - **binarySearch**: busca binária (sobrecarregado com versões genéricas);
  - **equals**: compara vetores de elementos primitivos e objetos da classe Object;
  - **fill**: preenche o vetor com valores de tipos primitivos e objetos da classe Object.

UsingArrays.java

```
import java.util.Arrays;

public class UsingArrays {
   private int intArray[] = { 1, 2, 3, 4, 5, 6 };
   private double doubleArray[] = { 8.4, 9.3, 0.2, 7.9, 3.4 };
   private int filledIntArray[], intArrayCopy[];

   public UsingArrays()  {
      filledIntArray = new int[ 10 ];
      intArrayCopy = new int[ intArray.length ];

      Arrays.fill( filledIntArray, 7 ); // preenche com 7s
      Arrays.sort( doubleArray ); // ordena crescentemente

      // preenche os vetores
      System.arraycopy( intArray, 0, intArrayCopy, 0, intArray.length )
          ;
   }
```

UsingArrays.java (cont.)

```
public void printArrays()  {
  System.out.print( "doubleArray: " );
  for ( double doubleValue : doubleArray )
     System.out.printf( "%.1f ", doubleValue );

  System.out.print( "\nintArray: " );
  for ( int intValue : intArray )
     System.out.printf( "%d ", intValue );

  System.out.print( "\nfilledIntArray: " );
  System.out.println(Arrays.toString(filledIntArray));

  System.out.print( "\nintArrayCopy: " );
  for ( int intValue : intArrayCopy )
     System.out.printf( "%d ", intValue );

  System.out.println( "\n" );
}
```

UsingArrays.java (cont.)

```
// pesquisa um valor no vetor
public int searchForInt( int value )
{
   return Arrays.binarySearch( intArray, value );
}

// compara o conteudo dos vetores
public void printEquality()
{
   boolean b = Arrays.equals( intArray, intArrayCopy );
   System.out.printf( "intArray %s intArrayCopy\n", ( b ? "==" : "!=
      " ) );

   b = Arrays.equals( intArray, filledIntArray );
   System.out.printf("intArray %s filledIntArray\n", ( b ? "==" : "
      !=" ) );
}
```

UsingArrays.java (cont.)

```
    public static void main( String args[] ) {
        UsingArrays usingArrays = new UsingArrays();
        usingArrays.printArrays();
        usingArrays.printEquality();
        int location = usingArrays.searchForInt( 5 );
        if ( location >= 0 )
            System.out.printf( "Found 5 at element %d in intArray\n",
                location );
        else
            System.out.println("5 not found in intArray");

        location = usingArrays.searchForInt( 8763 );
        if ( location >= 0 )
            System.out.printf( "Found 8763 at element %d in intArray\n",
                location );
        else
            System.out.println( "8763 not found in intArray" );
    }
}
```

UsingArrays.java (cont.)

```
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
```

Interface *Collection* e Classe *Collections*

- A **interface Collection** é a interface-raiz da hierarquia de todas interfaces de coleções
  - Contém operações realizadas em coleções inteiras (*bulk operations*)
    - Adicionar elementos;
    - Esvaziar;
    - Comparar.

Interface *Collection* e Classe *Collections* (cont.)

- Também contém operações que retornam iteradores (objetos *Iterator*) que nos permitem percorrer uma coleção.

- Comumente utilizada como um tipo de parâmetro nos métodos para permitir processamento polimórfico

Interface *Collection* e Classe *Collections* (cont.)

- A **classe Collections** fornece métodos que manipulam coleções polimorficamente
  - Implementam algoritmos para pesquisa e ordenação, entre outros;
  - Também fornece métodos adaptadores
  - Permitem que uma coleção seja tratada como sincronizada ou imutável

Listas

- A interface é implementada por diversas classes
  - Incluindo **ArrayList, LinkedList** e **Vector**
  - Novamente, ocorre *autoboxing* quando adicionamos elementos de tipos primitivos a estas coleções.

Listas (cont.)

- *Arraylists* se comportam como os *Vectors*
  - No entanto, não são sincronizados
  - Mais rápidos.
  - Podem ser utilizados para criar pilhas, filas, árvores e deques
  - O *Collections Framework* fornece algumas implementações destas estruturas

Métodos *ArrayList*

- boolean add(E e): adiciona um elemento no final da lista
- void add(int index, E element): adiciona um elemento em uma posição específica
- E remove(int index): remove um elemento de uma posição específica
- boolean remove(Object o): remove a primeira ocorrência de um elemento específico
- void clear(): remove todos os elementos
- boolean addAll(Collection <? extends E> c): adiciona todos os elementos de o uma coleção no final da lista

Métodos *ArrayList* (cont.)

- boolean addAll(int index, Collection <? extends E> c): insere todos os elementos de uma coleção na lista, começando na posição especificada.
- int size (): retorna o número de elementos na lista
- boolean isEmpty(): retorna verdadeiro se a lista está vazia
- boolean contains(Object o): retorna verdadeiro se o elemento *o* está na lista.
- int indexOf(Object o): retorna a posição da primeira ocorrência de *o*.

Métodos *ArrayList* (cont.)

- `int lastIndexOf(Object o)`: retorna a posição da última ocorrência
- `Object clone()`: cria uma copia rasa.
- `Object[] toArray()`: retorna um vetor com todos os elementos da lista
- `E get(int index)`: retorna o elemento especificado na posição
- `E set(int index,E element)`: substitui o elemento na posição *index*. Retorna o antigo elemento.
- `Iterator <E> iterator()`: retorna um iterador sobre os elementos da lista

Métodos *Iterator*

- boolean hasNext(): retorna verdadeiro se ainda existem elementos a ser percorridos
- E next(): lê e retorna o próximo elemento
- void remove(): remove o último elemento lido pelo iterador. Somente pode ser chamado uma única vez depois de cada chamada a next()

ArrayList e *Iterator*

- O exemplo a seguir demonstra vários recursos da interface *Collection*
  - O programa insere dois vetores de objetos *String* em dois *ArrayLists*
  - Usa iteradores para remover do primeiro *ArrayList* os elementos contidos no segundo *ArrayList*

ArrayList e *Iterator* (cont.)

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

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

   // cria um ArrayList, adiciona cores e manipula
   public CollectionTest() {
      List<String> list = new ArrayList<>();
      List<String> removeList =new ArrayList<>();

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

ArrayList e *Iterator* (cont.)

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

      System.out.println( "ArrayList: " );
      // exibe o conteudo
      for (int count = 0; count < list.size(); count++)
         System.out.printf( "%s ", list.get( count ) );

      // remove elementos
      removeColorMethod( list, removeList );

      System.out.println( "\n\nArrayList after calling removeColors: "
          );
      // exibe o conteudo
      for ( String color : list )
         System.out.printf( "%s ", color );
   }
```

ArrayList e *Iterator* (cont.)

```
    // remove elementos especificados em collection2 de collection1
    private void removeColorMethod(
          Collection< String > collection1,
          Collection< String > collection2 ) {
       // retorna o iterador
       Iterator< String > iterator = collection1.iterator();

       // percorre a colecao enquanto houverem itens
       while ( iterator.hasNext() )
          if ( collection2.contains( iterator.next() ) )
             iterator.remove(); // remove
    }

    public static void main( String args[] ) {
       new CollectionTest();
    }
}
```

ArrayList e *Iterator* (cont.)

```
ArrayList:
MAGENTA RED WHITE BLUE CYAN

ArrayList after calling removeColors:
MAGENTA CYAN
```

ArrayList e *Iterator* (cont.)

- Note que *ArrayList* é uma classe genérica do Java
  - Podemos especificar o tipo dos elementos como argumento
- A classe **Iterator** também é genérica
  - O método **hasNext** determina se há um próximo elemento na coleção;
  - O método **next** obtém uma referência a este próximo elemento;
  - O método **remove** apaga o elemento da coleção.

ArrayList e *Iterator* (cont.)

- **Se uma coleção é alterada por um método próprio depois de ter sido criado um iterador, o mesmo se torna inválido**
  - Qualquer operação com o iterador gerará uma *ConcurrentModificationException*

ArrayList e *Iterator* (cont.)

```
ArrayList<String> arr = new ArrayList<String>();
Iterator it = arr.iterator();
arr.add("bom");
arr.add("dia");
arr.add("hoje");
while(it.hasNext()){
    System.out.println(it.next());
} // gera erro
```

```
ArrayList<String> arr = new ArrayList<String>();
arr.add("bom");
arr.add("dia");
arr.add("hoje");
Iterator it = arr.iterator();
while(it.hasNext()){
    System.out.println(it.next());
}
```

LinkedList

- O exemplo a seguir demonstra operações em **LinkedLists**
  - O programa cria duas *LinkedLists* que contém Strings;
  - Os elementos de uma são adicionados à outra;
  - Então todas as *Strings* são convertidas para letras maiúsculas, e um intervalo destes elementos é removido.

LinkedList (cont.)

- ListIterator: Um iterador para listas que permite ao programador percorrer a lista em qualquer direção
  - boolean hasNext(): retorna verdadeiro se ainda ha elementos a ser percorridos
  - E next(): lê e retorna o próximo elemento da lista
  - boolean hasPrevious(): retorna verdadeiro se ainda existem elementos a ser lidos em sentido contrario
  - E previous(): lê e retorna o elemento previo
  - void remove(): Remove da lista o último elemento que foi retornado por next() ou previous(). Somente pode ser chamada uma única vez depois de cada next() ou previous()

LinkedList (cont.)

- void set(E e): substitui o último elemento retornado por next() ou previous(). Esta chamada pode ser feita somente se nem ListIterator .remove(), nem ListIterator .add() tiverem sido chamados após a última chamada para next() ou previous()
- void add(E e): insere um elemento na lista antes do próximo elemento retornado por next() ou depois de próximo elemento retornado por previous()

LinkedList (cont.)

```
import java.util.List;
import java.util.LinkedList;
import java.util.ListIterator;

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

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

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

LinkedList (cont.)

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

        list1.addAll( list2 ); // concatena as listas
        list2 = null; // libera
        printList( list1 ); // exibe os elementos
         // converte para maiusculas
        convertToUppercaseStrings( list1 );
        printList( list1 ); // exibe os elementos

        System.out.print("\nDeleting elements 4 to 7");
        removeItems( list1, 4, 7 ); // remove os itens 4-7 da lista
        printList( list1 ); // exibe os elementos
        // exibe os elementos na ordem inversa
        printReversedList( list1 );
    }
```

LinkedList (cont.)

```
// exibe os elementos da lista
public void printList( List< String > list ) {
   System.out.println( "\nlist: " );

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

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

LinkedList (cont.)

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

   while ( iterator.hasNext() ) {
      String color = iterator.next();   // retorna o item
      iterator.set( color.toUpperCase() ); // converte
   }
}

// obtem a sublista e a deleta usando o metodo clear
private void removeItems( List< String > list,
      int start, int end) {
   list.subList( start, end ).clear();   // remove os itens
}
```

LinkedList (cont.)

```
    // imprime a lista invertida
    private void printReversedList( List< String > list ) {
        ListIterator< String > iterator =
                list.listIterator( list.size() );

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

        // imprime a lista invertida
        while ( iterator.hasPrevious() )
            System.out.printf("%s ",iterator.previous() );
    }

    public static void main( String args[] )
    {
        new ListTest();
    }
}
```

LinkedList (cont.)

```
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 7...
list:
BLACK YELLOW GREEN BLUE WHITE BROWN BLUE GRAY SILVER

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

LinkedList (cont.)

- O método **AddAll** adiciona todos os elementos ao final da lista;
- O **método listIterator** retorna um iterador bidirecional
  - A **classe ListIterator** é uma classe genérica.
- O **método set** substitui um elemento da coleção por outro;

LinkedList (cont.)

- Os iteradores também possuem **métodos hasPrevious** e **previous**
  - Determina se há algum elemento anterior e retorna este elemento, respectivamente.
- Na classe *List*, o **método sublist** obtém um intervalo de valores contidos na lista original
  - Os parâmetros são o início e o final do intervalo, sendo que o final não está incluído entre os valores.

LinkedList (cont.)

- A classe *Arrays* fornece o método estático **asList** que permite ver um vetor como uma coleção *List*
  - Que encapsula o comportamento similar ao de uma lista encadeada
- O exemplo a seguir demonstra como criar uma *LinkedList* a partir de um vetor visto como uma *List*

UsingToArray.java

```
import java.util.LinkedList;
import java.util.Arrays;
import java.util.Iterator;

public class UsingToArray
{
   // cria uma LinkedList, adiciona elementos e converte para um vetor
   public UsingToArray()
   {
      String colors[] = { "black", "blue", "yellow" };

      LinkedList<String> links =
        new LinkedList< String >(Arrays.asList(colors));

      links.addLast( "red" );    // adiciona o ultimo item
      links.add( "pink" );       // adiciona ao final
      links.add( 3, "green" );   // adiciona no indice 3
      links.addFirst( "cyan" ); // adiciona como primeiro item
```

UsingToArray.java (cont.)

```
        // converte para um vetor
        colors = links.toArray(new String[links.size()]);

        System.out.println( "colors: " );
        for (String color: colors){
            System.out.printf("%s ", color);
        }

        System.out.println("\nlinks:");
        Iterator it = links.iterator();
        while(it.hasNext()){
            System.out.printf("%s ", it.next());
        }
    }
```

UsingToArray.java (cont.)

```
    public static void main( String args[] )
    {
        new UsingToArray();
    }
}
```

UsingToArray.java (cont.)

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

UsingToArray.java (cont.)

- Uma vez que obtemos uma *List* criada pelo método *asList* (sem alocar memória), o único método de modificação que podemos utilizar é o *set*
  - Qualquer outra tentativa de alteração gera **UnsupportedOperationException**
  - Como criamos uma *LinkedList* a partir do retorno do método *asList*, podemos alterá-la.

```
List<String> links2 =  Arrays.asList(colors);
links2.set(2, "gray"); // lança a exceção UnssupportedOperationException
```

UsingToArray.java (cont.)

- A partir de uma *List* também podemos obter um vetor com os mesmos elementos
  - Método *toArray()*

Vector

- Assim como *ArrayLists*, os **Vectors** fornecem uma estrutura parecida com um vetor
  - Pode se redimensionar automaticamente
  - Embora o comportamento sejam similares, os *Vectors* são sincronizados
  - Permitem operações que se valem do paralelismo de processamento.
- Vários dos métodos dos *Vectors* são demonstrados no exemplo a seguir.

VectorTest.java

```
import java.util.Vector;
import java.util.NoSuchElementException;

public class VectorTest
{
   private static final String colors[] = { "red", "white", "blue" };

   public VectorTest()
   {
      Vector< String > vector = new Vector< String >();
      printVector( vector );

      // adiciona elementos
      for ( String color : colors )
         vector.add( color );

      printVector( vector );
```

VectorTest.java (cont.)

```
// imprime o primeiro e o ultimo elementos
try {
   System.out.printf( "First element: %s\n", vector.firstElement
       ());
   System.out.printf( "Last element: %s\n", vector.lastElement()
       );
}
catch ( NoSuchElementException exception ) {
   exception.printStackTrace();
}

// testa se o vetor contem red"
if ( vector.contains( "red" ) )
 System.out.printf("\n\"red\" found at index %d\n\n", vector.
     indexOf("red"));
else
   System.out.println( "\n\"red\" not found\n" );
```

VectorTest.java (cont.)

```
    vector.remove( "red" ); // remove a string red"
    System.out.println( "\"red\" has been removed" );
    printVector( vector ); //

    // testa se o vetor contem red"depois da remocao
    if ( vector.contains( "red" ) )
       System.out.printf("\"red\" found at index %d\n", vector.
           indexOf("red"));
    else
       System.out.println( "\"red\" not found" );
    System.out.printf( "\nSize: %d\nCapacity: %d\n", vector.size(),
        vector.capacity() );
}
```

VectorTest.java (cont.)

```
   private void printVector( Vector< String > vectorToOutput )  {
      if ( vectorToOutput.isEmpty() )
         System.out.print( "vector is empty" );
      else  {// itera pelos elementos
         System.out.print( "vector contains: " );
         //exibe os elementos
         for ( String element : vectorToOutput )
            System.out.printf( "%s ", element );
      }

      System.out.println( "\n" );
   }

   public static void main( String args[] ) {
      new VectorTest();
   }
}
```

VectorTest.java (cont.)

```
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
```

Pilhas

- A classe **Stack** estende a classe Vector para implementar a estrutura de dados pilha
  - Ocorre *autoboxing* quando adicionamos um tipo primitivo a uma *Stack*
  - Só armazena referências a objetos.
- Métodos:
  - E push(E item): insere um elemento no topo da pilha
  - E pop(): remove o objeto que se encontra no topo da pilha
  - E peek(): lê o elemento que se encontra no topo sem removê-lo.
  - boolean empty(): verifica se a pilha está vazia.

Pilhas (cont.)

- O exemplo a seguir demonstra vários métodos da classe *Stack*

Pilhas (cont.)

```
import java.util.Stack;
import java.util.EmptyStackException;

public class StackTest
{
   public StackTest()
   {
      Stack< Number > stack = new Stack< Number >();

      // cria os numeros a serem armazenados na pilha
      Long longNumber = 12L;
      Integer intNumber = 34567;
      Float floatNumber = 1.0F;
      Double doubleNumber = 1234.5678;
```

Pilhas (cont.)

```
// usa o metodo push
stack.push( longNumber );
printStack( stack );
stack.push( intNumber );
printStack( stack );
stack.push( floatNumber );
printStack( stack );
stack.push( doubleNumber );
printStack( stack );
// remove os itens da pilha
```

Pilhas (cont.)

```
        try
        {
            Number removedObject = null;

            while ( true )
            {
                removedObject = stack.pop();
                System.out.printf( "%s popped\n", removedObject );
                printStack( stack );
            }
        }
        catch ( EmptyStackException emptyStackException )
        {
            emptyStackException.printStackTrace();
        }
    }
```

Pilhas (cont.)

```
    private void printStack( Stack< Number > stack ) {
        if ( stack.isEmpty() )
            System.out.print( "stack is empty\n\n" ); // pilha vazia
        else {
            System.out.print( "stack contains: " );

            // itera atraves dos elementos
            for ( Number number : stack )
                System.out.printf( "%s ", number );

            System.out.print( "(top) \n\n" ); // indica o topo da pilha
        }
    }

    public static void main( String args[] ) {
        new StackTest();
    }
}
```

Pilhas (cont.)

```
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)
```

Pilhas (cont.)

```
34567 popped
stack contains: 12 (top)

12 popped

stack is empty
java.util.EmptyStackException
at java.util.Stack.peek(Stack.java:102)
at java.util.Stack.pop(Stack.java:84)
at javastack.JavaStack.<init>(JavaStack.java:33)
at javastack.JavaStack.main(JavaStack.java:55)
```

Filas de Prioridade

- A interface *Queue* estende *Collection* e adiciona novos métodos para inserir, remover e inspecionar elementos de uma fila
  - A classe **PriorityQueue** implementa esta interface e ordena os elementos de acordo com o método *compareTo* (*Comparable*) ou um objeto *Comparator*
  - As inserções são ordenadas e as remoções são realizadas no início da estrutura
    - O primeiro elemento é o de maior prioridade.

Filas de Prioridade (cont.)

- As operações mais comuns são:
  - **offer**: insere um elemento na posição apropriada de acordo com sua prioridade;
  - **poll**: remove o elemento de maior prioridade;
  - **peek**: retorna uma referência ao objeto de maior prioridade, sem removê-lo;
  - **clear**: remove todos os elementos;
  - **size**: retorna o número de elementos.

PriorityQueueTest.java

```
import java.util.PriorityQueue;

public class PriorityQueueTest {
   public static void main( String args[] ) {
      // fila de capacidade 11
      PriorityQueue< Double > queue = new PriorityQueue< Double >();

      // insere os elementos na fila
      queue.offer( 3.2 );
      queue.offer( 9.8 );
      queue.offer( 5.4 );
```

PriorityQueueTest.java (cont.)

```
        System.out.print( "Polling from queue: " );

        // exibe os elementos da fila
        while ( queue.size() > 0 )
        {
            System.out.printf( "%.1f ", queue.peek() ); // exibe o elemento do
                topo
            queue.poll(); // remove o elemento do topo
        }
    }
}
```

PriorityQueueTest.java (cont.)

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

Conjuntos (*Set*)

- O *Java Collections Framework* possui diversas implementações da interface **Set**, incluindo
  - **HashSet**: armazena os elementos em uma tabela hash;
  - **TreeSet**: armazena os elementos em uma árvore.
  - **LinkedHashSet**: armazena os elementos em uma tabela hash com listas encadeadas.

Conjuntos (*Set*) (cont.)

- Uma interface interessante que também implementa a interface **Set**:
  - **SortedSet**: mantém os elementos ordenados, seja pela ordem natural dos tipos primitivos, seja pelo uso de comparadores.
- O exemplo a seguir utiliza *HashSet* para remover *Strings* duplicadas de uma *List*

SetTest.java

```
import java.util.List;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;
import java.util.Collection;

public class SetTest
{
   private static final String colors[] = { "red", "white", "blue", "
      green", "gray", "orange", "tan", "white", "cyan", "peach", "gray"
      , "orange" };

   // cria e exibe o ArrayList
   public SetTest() {
      List< String > list = Arrays.asList( colors );
      System.out.printf( "ArrayList: %s\n", list );
      printNonDuplicates( list );
   }
```

SetTest.java (cont.)

```
    //cria o conjunto a partir do vetor, para eliminar duplicatas
    private void printNonDuplicates( Collection< String > collection )
        {
        // cria o HashSet
        Set< String > set = new HashSet< String >( collection );

        System.out.println( "\nNonduplicates are: " );

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

        System.out.println();
    }

    public static void main( String args[] ) {
        new SetTest();
    }
}
```

SetTest.java (cont.)

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

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

Set - exemplo

```
public class Dog implements Comparable<Dog> {
    int size;

    public Dog(int size) {
        this.size = size;
    }
    @Override
    public String toString() {
        return "Dog(" + size + ")";
    }
    @Override
    public int compareTo(Dog o){
        return size - o.size;
    }
}
```

Set - exemplo (cont.)

```
import java.util.Collection;
import java.util.HashSet;
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.Set;
import java.util.TreeSet;

public class SetDriver {
    Set<Dog> hashset, treeset, linkedhashset;

    public SetDriver() {
        hashset = new HashSet<>();
        treeset = new TreeSet<>();
        linkedhashset = new LinkedHashSet<>();
    }
```

Set - exemplo (cont.)

```
public void fillSets(){
    hashset.add(new Dog(2));
    hashset.add(new Dog(1));
    hashset.add(new Dog(3));
    hashset.add(new Dog(5));
    hashset.add(new Dog(4));

    linkedhashset.add(new Dog(2));
    linkedhashset.add(new Dog(1));
    linkedhashset.add(new Dog(3));
    linkedhashset.add(new Dog(5));
    linkedhashset.add(new Dog(4));

    treeset.add(new Dog(2));
    treeset.add(new Dog(1));
    treeset.add(new Dog(3));
    treeset.add(new Dog(5));
    treeset.add(new Dog(4));
}
```

Set - exemplo (cont.)

```
    private void print( Collection<Dog> collection ){
        Iterator<Dog> it = collection.iterator();
        while ( it.hasNext() ){
            System.out.print(it.next() + " ");
        }
        System.out.println("");
    }

    public void showSets(){
        System.out.println("Hash set");
        print(hashset);
        System.out.println("LinkedHash set");
        print(linkedhashset);
        System.out.println("Tree set");
        print(treeset);
    }

}
```

Set - exemplo (cont.)

```
public class TestSets {

    public static void main(String[] args) {
        // TODO code application logic here
        SetDriver s = new SetDriver();
        s.fillSets();
        s.showSets();
    }

}
```

Set - exemplo (cont.)

```
Hash set
Dog(2) Dog(4) Dog(3) Dog(1) Dog(5)
LinkedHash set
Dog(2) Dog(1) Dog(3) Dog(5) Dog(4)
Tree set
Dog(1) Dog(2) Dog(3) Dog(4) Dog(5)
```

Mapas

- Três das várias classes que implementam a interface Map são:
  - **Hashtable** e **HashMap**: armazenam os elementos em tabelas *hash*
  - **TreeMap**: armazenam os elementos em árvores.

Mapas (cont.)

- Algumas interfaces interessantes que também implementam a interface *Map* incluem:
  - **MultiMap**: permite uma coleção de valores para uma mesma chave;
  - **SortedMap**: mantém os elementos ordenados, seja pela ordem natural dos tipos primitivos, seja pelo uso de comparadores.

Mapas (cont.)

- Métodos:
  - boolean containsKey(Object key): retorna verdadeiro se o map contem a chave
  - boolean containsValue(Object value): retorna verdadeiro se o map contem chave(s) com o valor especificado
  - V get(Object key): retorna o valor
  - V put(K key, V value): substitui o valor asociado a uma chave
  - Set<K> keySet(): retorna um Set das chaves
  - Collection <V> values(): retorna uma coleção dos valores
- O exemplo a seguir utiliza *HashMap* para contar o número de ocorrências de palavras em uma *String*

WordTypeCount.java

```
import java.util.StringTokenizer;
import java.util.Map;
import java.util.HashMap;
import java.util.Set;
import java.util.TreeSet;
import java.util.Scanner;

public class WordTypeCount {
   private Map< String, Integer > map;
   private Scanner scanner;

   public WordTypeCount() {
      // cria o HashMap
      map = new HashMap< String, Integer >();
      scanner = new Scanner( System.in );
      createMap(); // cria o mapa baseado na entrada
      displayMap(); // exibe o conteúdo do mapa
   }
```

WordTypeCount.java (cont.)

```
private void createMap() {
   System.out.println( "Enter a string:" );
   String input = scanner.nextLine();

   // cria um StringTokenizer para a entrada
   StringTokenizer tokenizer = new StringTokenizer( input );
```

WordTypeCount.java (cont.)

```
        // processa o texto da entrada
        while ( tokenizer.hasMoreTokens() ) // enquanto houver entrada
        {
            String word = tokenizer.nextToken().toLowerCase(); // pega a
                palavra

            // se o mapa contem a palavra
            if ( map.containsKey( word ) ) {
                int count = map.get( word ); // retorna a contagem atual
                map.put( word, count + 1 );  // incrementa a contagem
            }
            else
                map.put( word, 1 ); // adiciona uma nova palavra com o contador valendo 1
        }
    }
```

WordTypeCount.java (cont.)

```
private void displayMap() {
   Set< String > keys = map.keySet(); // obtem as chaves

   // ordena as chaves
   TreeSet< String > sortedKeys = new TreeSet< String >( keys );

   System.out.println( "Map contains:\nKey\t\tValue" );

   // gera a saída para cada chave no mapa
   for ( String key : sortedKeys )
      System.out.printf( "%-10s%10s\n", key, map.get( key ) );

   System.out.printf("\nsize:%d\nisEmpty:%b\n", map.size(), map.
       isEmpty());
}
```

WordTypeCount.java (cont.)

```
    public static void main( String args[] ){
        new WordTypeCount();
    }
}
```

WordTypeCount.java (cont.)

```
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
```

Mapas

- Um **StringTokenizer** quebra uma string em palavras individuais
  - Determinadas por espaços em branco;
  - O método *hasMoreTokens* determina se ainda há palavras a serem processadas;
  - O método *nextToken* retorna o token em uma String.

Mapas (cont.)

- Outros métodos utilizados incluem:
  - **containsKey**: determina se a chave está contida no mapa
  - **put**: cria uma nova entrada chave/valor no mapa;
  - **get**: obtém o valor associado a uma chave;
  - **keySet**: retorna o conjunto de chaves do mapa;
  - **size**: retorna a quantidade de pares chave/valor do mapa;
  - **isEmpty**: retorna true ou false para indicar se o mapa está vazio

Algoritmos

- O *Java Collections Framework* fornece vários algoritmos de alta performance para manipular elementos de uma coleção
  - Alguns operam em *Lists*, outros em *Collections*
  - Todos os algoritmos são polimórficos

Algoritmos (cont.)

- Alguns algoritmos utilizam um recurso chamado **comparador (comparator)**
  - Objeto de uma classe que implementa a interface **Comparator**, um tipo genérico que recebe um parâmetro
  - O método **compare** deve ser implementado
    - Retorna um valor positivo se o primeiro elemento for maior ou um valor negativo se o primeiro elemento for menor;
    - Caso contrário retorna zero.

Algoritmos (cont.)

Algoritmo	Descrição
sort	Ordena os elementos de um *List*.
binarySearch	Pesquisa um objeto de um *List*.
reverse	Inverte as posições dos objetos de um *List*.
shuffle	Embaralha os elementos de um *List*.
fill	Define que cada elemento de um *List* referencia um objeto especificado.
copy	Copia as referências de um *List* para outro.
min	Retorna o menor elemento de uma coleção.
max	Retorna o maior elemento de uma coleção.
addAll	Adiciona todos os elementos de um vetor a uma coleção.
frequency	Calcula quantos elementos de uma coleção são iguais ao elemento especificado.
disjoint	Determina se duas coleções não possuem elementos em comum.

Exemplo de ordenação

Seja a classe **Conta** que consta dos atributos *codigo* e *nome*. Gerar uma lista que contenha um conjunto de Contas, ordenar os dados baseados no código da conta usando a interface *Comparable* (a comparação é realizada entre o objeto que chama o método e o objeto passado como parâmetro)

Exemplo de ordenação (cont.)

```
public class Conta implements Comparable<Conta>{
    private int num;
    private String nome;
    private double saldo;

    public Conta(){
        this(0, "", 0.0);
    }

    public Conta(int num, String nome, double saldo){
        this.num = num;
        this.nome = nome;
        this.saldo = saldo;
    }
```

Exemplo de ordenação (cont.)

```
@Override
public int compareTo(Conta o){
    if (this.num < o.num)
        return -1;
    if (this.num > o.num)
        return 1;
    return 0;
}

@Override
public String toString(){
   return String.format("%d %s %f", num, nome, saldo);
}
```

Exemplo de ordenação (cont.)

```
    public String getNome(){
        return nome;
    }

    public double getSaldo() {
        return saldo;
    }
}
```

Exemplo de ordenação (cont.)

```
package javaordena;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class JavaOrdena {

    public static void print(String titulo, Collection<Conta>
        collection){
        System.out.println(titulo);
        for (Conta elem : collection)
            System.out.println(elem);
    }
```

Exemplo de ordenação (cont.)

```
public static void insere(Collection<Conta> collection){
    Conta obj = new Conta(266, "Marco", 500.45);
    collection.add(obj);
    Conta obj2 = new Conta(29, "Pedro", 1909.00);
    collection.add(obj2);
    Conta obj3 = new Conta(101, "Andre", 200.00);
    collection.add(obj3);
}
```

Exemplo de ordenação (cont.)

```
public static void main(String[] args) {
    // TODO code application logic here
    List<Conta> contas = new ArrayList<>();

    // insere contas na colecao
    insere(contas);

    // (1) ordena pelo codigo
    Collections.sort(contas);
    print("Ordena pelo numero de conta", contas);
    // comparator que ordena pelo nome
    Comparator<Conta> com = new Comparator<Conta>(){
        @Override
        public int compare(Conta c1, Conta c2){
            String nome1 = c1.getNome();
            String nome2 = c2.getNome();
            return nome1.compareTo(nome2);
        }
    };
```

Exemplo de ordenação (cont.)

```
        // (2) sort ordena pelo nome
        Collections.sort(contas, com);
        print("Ordena pelo nome de conta", contas);
        // (3)vsort oom comparator que ordena pelo saldo
        Collections.sort(contas, new Comparator(){
            @Override
            public int compare(Object o1, Object o2){
                double saldo1 = ((Conta)o1).getSaldo();
                double saldo2 = ((Conta)o2).getSaldo();
                if (saldo1 > saldo2) return 1;
                else if (saldo1 < saldo2) return -1;
                else return 0;
            }
        });
        print("Ordena pelo saldo de conta", contas);
    }
}
```

Exemplo de ordenação (cont.)

```
Ordena pelo numero de conta
29 Pedro 1909.000000
101 Andre 200.000000
266 Marco 500.450000

Ordena pelo nome de conta
101 Andre 200.000000
266 Marco 500.450000
29 Pedro 1909.000000

Ordena pelo saldo de conta
101 Andre 200.000000
266 Marco 500.450000
29 Pedro 1909.000000
```

Exemplo de ordenação

Seja a classe **Conta** que consta dos atributos *codigo* e *nome*. Gerar uma lista que contenha um conjunto de Contas, ordenar os dados baseados no nome usando a interface *Comparator* (a comparação é realizada entre dois objetos passados como parâmetro)

Exemplo de ordenação (cont.)

```
public class Conta{
    private int num;
    private String nome;
    public Conta(){
        this(0, "");
    }
    public Conta(int num, String nome){
        this.num = num;
        this.nome = nome;
    }
    public String toString(){
       return String.format("\n%d %s", num, nome);
    }
    public String getNome(){
        return nome;
    }
}
```

Exemplo de ordenação (cont.)

```
import java.util.Comparator;

public class SortbyName implements Comparator<Conta>{
    public int compare(Conta a, Conta b){
        return a.getNome().compareTo(b.getNome());
    }
}
```

Exemplo de ordenação (cont.)

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class JAvaOrdena {
    public static void main(String[] args) {
        List<Conta> conta = new ArrayList<>();
        Conta obj = new Conta(266, "Marco");
        conta.add(obj);
        Conta obj2 = new Conta(29, "Pedro");
        conta.add(obj2);
        Conta obj3 = new Conta(101, "Andre");
        conta.add(obj3);
```

Exemplo de ordenação (cont.)

```
        Collections.sort(conta, new SortbyName());
        for (Conta elem : conta)
            System.out.println(elem);
    }
}
```

Exemplo de ordenação (cont.)

```
101 Andre
266 Marco
29 Pedro
```

FIM