

# Java - Genéricos

BCC 221 - Programação Orientada a Objectos(POO)

Guillermo Cámara-Chávez

Departamento de Computação - UFOP



# Genéricos

- ▶ A programação de genéricos nos permite criar modelos genéricos
  - ▶ Métodos genéricos especificam em uma única declaração um **conjunto de métodos de relacionados**;
  - ▶ Classes genéricas especificam em uma única declaração um **conjunto de tipos relacionados**;

# Métodos Genéricos

- ▶ Se as **operações** realizadas por **diversos métodos sobrecarregados são idênticas** para todos os tipos de argumentos, tais métodos podem ser **mais convenientemente codificados**
  - ▶ Usando um método genérico;
  - ▶ Uma mesma declaração pode ser invocada com diferentes parâmetros;
  - ▶ O compilador trata cada chamada ao método de acordo com os parâmetros enviados.
- ▶ O exemplo a seguir imprime o conteúdo de vetores, não importando o tipo dos elementos

## Métodos Genéricos (cont.)

- ▶ A declaração de métodos genéricos começa com a **seção de parâmetro de tipo**, delimitado por `<` e `>`
  - ▶ Antes do tipo de retorno no método;
  - ▶ Cada seção contém um ou mais parâmetros de tipo, separados por vírgulas.

# Métodos Genéricos (cont.)

- ▶ Um parâmetro de tipo especifica o nome de um tipo genérico
  - ▶ Pode ser utilizado para o tipo de retorno, tipo dos parâmetros do método e também variáveis locais;
  - ▶ Age como uma reserva para os tipos verdadeiros

## Métodos Genéricos (cont.)

- ▶ O corpo de um método genérico é declarado de forma semelhante a um método comum;
- ▶ Parâmetros de tipo podem somente representar tipos de **referências**
  - ▶ Tipos primitivos como *int*, *double* e *char* não;
  - ▶ Todo os dados enviados como parâmetros devem ser objetos de classes ou interfaces

# Métodos Genéricos (cont.)

```
public class GenericMethodTest {  
    //metodo generico  
    public static < E > void printArray( E[] inputArray  
        ){  
        // exhibe os elementos do vetor  
        for ( E element : inputArray )  
            System.out.printf( "%s ", element );  
        System.out.println();  
    }  
  
    public static void main( String args[] ) {  
        // cria vetores dos tipos Integer, Double e Character  
        Integer[] integerArray = {1,2,3,4,5,6};  
        Double[] doubleArray = {1.1,2.2,3.3,4.4,5.5,6.6};  
        Character[] characterArray ={'H','E','L','L','O'};
```

## Métodos Genéricos (cont.)

```
System.out.println( "\nintegerArray contains:" );  
printArray( integerArray ); // envia um vetor de Integer  
System.out.println( "\ndoubleArray contains:" );  
printArray( doubleArray ); // envia um vetor de Double  
System.out.println( "\ncharacterArray contains:" );  
// envia um vetor de Character  
printArray( characterArray );  
}  
}
```



# Métodos Genéricos (cont.)

`integerArray` contains

`1 2 3 4 5 6`

`doubleArray` contains

`1.1 2.2 3.3 4.4 5.5 6.6`

`characterArray` contains

`H E L L O`

# Métodos Genéricos (cont.)

- ▶ No exemplo, declaramos o parâmetro de tipo **E**
  - ▶ Aparece também na lista de parâmetros e no for aprimorado;
  - ▶ Por padrão, o nome deve ser somente uma letra maiúscula.
- ▶ No *main*, diferentes vetores são passados para o método genérico
  - ▶ Inicialmente, o compilador procura uma versão do método específica para o parâmetro;
  - ▶ Não encontrando, a versão genérica é utilizada.

## Métodos Genéricos (cont.)

- ▶ Quando o compilador traduz o código para *bytecode*, os métodos genéricos têm seus argumentos substituídos por tipos de verdade
  - ▶ Por padrão, o tipo `Object` é utilizado;
  - ▶ Diferentemente do que ocorre em C++, em que uma cópia para cada tipo utilizado é criada.
- ▶ A seguir é apresentado o equivalente ao método genérico do código anterior depois de compilado

## Métodos Genéricos (cont.)

```
public static void printArray( Object[] inputArray )
{
    // exhibe os elementos do vetor
    for ( Object element : inputArray )
        System.out.printf( "%s ", element );

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

# Métodos Genéricos (cont.)

- ▶ Os métodos genéricos podem ser sobrecarregados
  - ▶ Por outros métodos genéricos;
  - ▶ Por métodos específicos
    - ▶ Inclusive tendo os mesmos parâmetros;
    - ▶ Têm precedência maior em relação ao genérico

# Classes Genéricas

- ▶ Os **conceitos de estruturas de dados**, como uma pilha, são **independentes dos tipos** dos elementos que elas manipulam
  - ▶ Desta forma, podemos **criar uma classe que descreva o comportamento** de uma estrutura de dados, de uma maneira independente
  - ▶ Ao instanciarmos esta **classe genérica**, podemos especificar qual é o tipo desejado;
  - ▶ **Esta capacidade permite** um grande avanço na **reusabilidade** de código.
- ▶ O tratamento dispensado pelo compilador às classes genéricas é semelhante ao dispensado aos métodos genéricos.

# Classes Genéricas (cont.)

- ▶ Estas classes são conhecidas como **classes parametrizadas**
  - ▶ Ou *tipos parametrizados*, uma vez que podem receber um ou mais parâmetros;
  - ▶ Tais parâmetros representam apenas tipos de referência
    - ▶ Ou seja, uma estrutura não poderia ser instanciada com um tipo **primitivo**;
    - ▶ No entanto, podemos utilizar o **autoboxing** para converter tipos primitivos em objetos

# Classes Genéricas (cont.)

- ▶ **Autoboxing** consiste em atribuir um tipo primitivo a uma variável ou estrutura cujo tipo é uma classe empacotadora (*wrapper class*)
  - ▶ A conversão é implícita;
  - ▶ Por exemplo, *int* para *Integer*
- ▶ O **Auto-Unboxing** é o processo contrário;



# Classes Genéricas (cont.)

- ▶ Em Java há 8 classes empacotadoras
  - ▶ *Byte, Short, Integer, Long, Float, Double, Character e Boolean*
  - ▶ Todos declaradas no pacote *java.lang*

# Classes Genéricas (cont.)

- ▶ O exemplo a seguir apresenta a declaração de uma classe que descreve uma pilha genérica
  - ▶ O parâmetro  $E$  representa o tipo dos elementos a serem manipulados pela pilha
    - ▶ Uma classe genérica pode possuir mais que um parâmetro de tipo, separados por vírgula.
  - ▶ Este parâmetro é utilizado ao longo do código nos trechos em que é necessário indicar o tipo dos elementos

# EmptyStackException.java

```
public class EmptyStackException extends
    RuntimeException{

    public EmptyStackException(){
        this("Stack is empty");
    }

    public EmptyStackException(String msg){
        super(msg);
    }
}
```

# FullStackException.java

```
public class FullStackException extends
    RuntimeException {

    public FullStackException(){
        this("Stack is full");
    }

    public FullStackException(String msg){
        super(msg);
    }
}
```

# Stack.java

```
public class Stack< E >
{
    private final int size; // numero de elementos da pilha
    private int top; // indice do topo
    private E[] elements; // vetor para armazenar os elementos

    // o tamanho padrao e 10
    public Stack()
    {
        this( 10 );
    }

    // constroi uma pilha com um tamanho especificado
    public Stack( int s )
    {
        size = s > 0 ? s : 10;
        top = -1; // pilha vazia inicialmente

        elements = ( E[] ) new Object[ size ]; // cria o vetor
    }
}
```

## Stack.java (cont.)

```
public void push( E pushValue )
{
    if ( top == size - 1 )
        throw new FullStackException( String.format( "
            Stack is full, cannot push %s", pushValue
        ) );

    elements[ ++top ] = pushValue;
}

public E pop()
{
    if ( top == -1 )
        throw new EmptyStackException( "Stack is empty
            , cannot pop" );

    return elements[ top-- ];
}
}
```

# Classes Genéricas

- ▶ O mecanismo dos genéricos em Java não permite que tipos passados por parâmetro sejam utilizados para a criação de vetores
  - ▶ É necessário criar um vetor de *Object* e realizar um cast;
  - ▶ Gera um warning “*unchecked cast*”;
  - ▶ No entanto, o compilador não pode garantir totalmente que o vetor nunca conterá objetos de outros tipos que não sejam o passado como parâmetro.

# Classes Genéricas (cont.)

- ▶ O escopo do parâmetro de tipo de uma classe genérica é a classe inteira
  - ▶ No entanto, os parâmetros não podem ser utilizados em declarações do tipo static.



## Classes Genéricas (cont.)

- ▶ Vejamos como exemplo uma aplicação que utiliza métodos genéricos para testar a classe genérica vista anteriormente
  - ▶ Para os tipos *Integer* e *Double*;
  - ▶ Os métodos que testam os métodos *push* e *pop* também recebem como parâmetro o tipo a ser utilizado nos testes;
  - ▶ Elementos destes tipos são enviados aos métodos *push* e retornados pelo método *pop*
  - ▶ Note a utilização de classes adaptadoras.

# StackTest.java

```
public class StackTest {  
    private double[] doubleElements={1.1, 2.2, 3.3, 4.4,  
        5.5, 6.6};  
    private int[] integerElements = { 1, 2, 3, 4, 5, 6,  
        7, 8, 9, 10, 11 };  
  
    private Stack< Double > doubleStack; // pilha de Double  
    private Stack< Integer > integerStack; // pilha de Integer  
  
    public void testStacks() {  
        doubleStack = new Stack< Double >( 5 );  
        integerStack = new Stack< Integer >( 10 );  
  
        testPush("doubleStack", doubleStack ,  
            doubleElements);  
        testPop("doubleStack", doubleStack);  
        testPush("integerStack", integerStack ,  
            integerElements);  
        testPop("integerStack", integerStack);  
    }  
}
```

## StackTest.java (cont.)

```
// metodo generico que testa o metodo push da classe generica
public < T > void testPush( String name,
    Stack< T > stack , T[] elements ) {
    try {
        System.out.printf( "\nPushing elements onto %s
            \n", name );

        for ( T element : elements ) {
            System.out.printf( "%s ", element );
            stack.push( element );
        }
    }
    catch ( FullStackException fullStackException )
    {
        System.out.println();
        fullStackException.printStackTrace();
    }
}
```

## StackTest.java (cont.)

```
// metodo generico que testa o metodo pop da classe generica
public < T > void testPop( String name, Stack< T >
    stack ) {
    try {
        System.out.printf( "\nPopping elements from %s
            \n", name );
        T popValue;

        while ( true ) {
            popValue = stack.pop();
            System.out.printf( "%s ", popValue );
        }
    }
    catch( EmptyStackException emptyStackException ){
        System.out.println();
        emptyStackException.printStackTrace();
    }
}
```

## StackTest.java (cont.)

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

## StackTest.java (cont.)

Pushing elements onto doubleStack

1.1 2.2 3.3 4.4 5.5 6.6

```
javastackgenerico.FullStackException: Stack is full ,  
    cannot push 6.6
```

```
at javastackgenerico.Stack.push(Stack.java:20)
```

```
at javastackgenerico.JavaStackGenerico.testPush(  
    JavaStackGenerico.java:36)
```

```
at javastackgenerico.JavaStackGenerico.testStacks(  
    JavaStackGenerico.java:21)
```

```
at javastackgenerico.JavaStackGenerico.main(  
    JavaStackGenerico.java:69)
```

## StackTest.java (cont.)

Popping elements from doubleStack

5.5 4.4 3.3 2.2 1.1

javastackgenerico.EmptyStackException: Stack is empty,  
cannot pop

at javastackgenerico.Stack.pop(Stack.java:26)

at javastackgenerico.JavaStackGenerico.testPop(  
JavaStackGenerico.java:56)

at javastackgenerico.JavaStackGenerico.testStacks(  
JavaStackGenerico.java:22)

at javastackgenerico.JavaStackGenerico.main(  
JavaStackGenerico.java:69)

## StackTest.java (cont.)

Pushing elements onto integerStack

1 2 3 4 5 6 7 8 9 10 11

javastackgenerico.FullStackException: Stack is full ,  
cannot push 11

at javastackgenerico.Stack.push(Stack.java:20)

at javastackgenerico.JavaStackGenerico.testPush(  
JavaStackGenerico.java:36)

at javastackgenerico.JavaStackGenerico.testStacks(  
JavaStackGenerico.java:23)

at javastackgenerico.JavaStackGenerico.main(  
JavaStackGenerico.java:69)



## StackTest.java (cont.)

Popping elements from integerStack

10 9 8 7 6 5 4 3 2 1

javastackgenerico.EmptyStackException: Stack is empty,  
cannot pop

at javastackgenerico.Stack.pop(Stack.java:26)

at javastackgenerico.JavaStackGenerico.testPop(  
JavaStackGenerico.java:56)

at javastackgenerico.JavaStackGenerico.testStacks(  
JavaStackGenerico.java:24)

at javastackgenerico.JavaStackGenerico.main(  
JavaStackGenerico.java:69)

# Tipos “Cru”

- ▶ Os exemplos anteriores instanciavam a classe *Stack* com tipos *Integer* e *Double* passados por argumento
  - ▶ Também é possível instanciar uma classe genérica sem especificar o tipo, como a seguir:

```
// nenhum tipo especificado  
Stack objectStack = new Stack( 5 );
```

- ▶ Neste caso, dizemos que o objeto possui um tipo “cru” (*raw type*);
- ▶ O compilador utiliza o tipo *Object* implicitamente, criando uma pilha para qualquer tipo de elemento;
- ▶ Há insegurança quanto ao tipo dos dados armazenados em um tipo cru

## Tipos “Crus” (cont.)

- ▶ Os tipos crus são importante para a compatibilidade das versões antigas do Java
- ▶ É possível atribuir uma estrutura de tipo cru a uma estrutura que especifique o tipo, como abaixo:

```
Stack rawTypeStack2 = new Stack< Double >( 5 );
```

# Coringas em Métodos Genéricos

- ▶ Quando não pudermos determinar a classe específica dos elementos que serão passados a um genérico, podemos utilizar um **coringa (wildcard)**
  - ▶ Por exemplo, em um método que soma os elementos de um vetor, podemos não saber se tais elementos serão dos tipos *Integer* ou *Double*
  - ▶ Podemos então indicar simplesmente que o tipo será o de uma classe que estende a classe *Number*
    - ▶ De fato, *Integer* e *Double* são subclasses de *Number*
  - ▶ Um parâmetro coringa é indicado por uma **?**, como abaixo:  
`ArrayList< ? extends Number > list`

## Coringas em Métodos Genéricos (cont.)

```
public static double sum( ArrayList< ? extends
    Number > list )
{
    double total = 0;

    for ( Number element : list )
        total += element.doubleValue();

    return total;
}
```

## Coringas em Métodos Genéricos (cont.)

- ▶ Uma desvantagem desta sintaxe é que o símbolo ? não pode ser utilizado como o nome de um tipo ao longo do método
  - ▶ Por exemplo, no for aprimorado não podemos substituir Number por ?
- ▶ Uma alternativa é declarar o método como a seguir:

```
public static <T extends Number> double sum(  
    ArrayList< T > list )
```

# Genéricos e Herança

- ▶ Genéricos podem ser utilizados com herança em diversas maneiras:
  - ▶ Uma classe genérica pode ser derivada de uma classe não genérica;
  - ▶ Uma classe genérica pode ser derivada a partir de outra;
  - ▶ Uma classe não genérica pode ser derivada a partir de uma classe genérica;
  - ▶ Um método genérico em uma subclasse pode sobrescrever um método genérico da superclasse, se as assinaturas forem idênticas.

# 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
<i>Collection</i>	A classe raiz na hierarquia de coleções, a partir da qual todas as outras são derivadas.
<i>Set</i>	Uma coleção que não contém repetições.
<i>List</i>	Uma coleção ordenada que pode conter repetições.
<i>Map</i>	Associa chaves a valores e não pode conter chaves duplicadas.
<i>Queue</i>	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 `List` 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 dequeues
  - ▶ O *Collections Framework* fornece algumas implementações destas estruturas

# 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<String >();
        List<String> removeList =new ArrayList<String>();

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

// remove elementos
removeColors( list , removeList );

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

## ArrayList e Iterator (cont.)

```
// remove elementos especificados em collection2 de collection1
private void removeColors(
    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`

## 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());  
}
```

```
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());  
}
```

- ▶ 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.)

```
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 ); // exhibe os elementos
// converte para maiúsculas
convertToUppercaseStrings( list1 );
printList( list1 ); // exhibe os elementos

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

## LinkedList (cont.)

```
// exhibe 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.
- ▶ 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

## *LinkedList* (cont.)

- ▶ 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;

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());
}

}

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");
```

## 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
```

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

# 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.
- ▶ 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 );
printStats( stack );
stack.push( intNumber );
printStats( stack );
stack.push( floatNumber );
printStats( stack );
stack.push( doubleNumber );
printStats( 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: " );

// exhibe os elementos da fila
while ( queue.size() > 0 )
{
    System.out.printf( "%.1f ", queue.peek() ); //
        exhibe 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.

# 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 exhibe 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
```

# 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.
- ▶ 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(); // exhibe 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
```

- ▶ 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

## 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;  
  
    public Conta(){  
        this(0, "");  
    }  
  
    public Conta(int num, String nome){  
        this.num = num;  
        this.nome = nome;  
    }  
}
```

## Exemplo de ordenação (cont.)

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

public String toString(){
    return String.format("\n%d %s", num, nome);
}

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

## 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);

        // ordena por código
        Collections.sort(conta);
        for (Conta elem : conta)
            System.out.println(elem);
    }
}
```

## Exemplo de ordenação (cont.)

```
// ordena por nome
Collections.sort(conta, new Comparator(){
    public int compare(Object o1, Object o2){
        String nome1 = ((Conta)o1).getNome();
        String nome2 = ((Conta)o2).getNome();
        return nome1.compareTo(nome2);
    }
})
}
```

## 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);

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



FIM