



BCC221

Programação Orientada a Objetos

Prof. Marco Antonio M. Carvalho
2014/2



UFOP

Endereços Importantes

- Site da disciplina:
<http://www.decom.ufop.br/marco/>
- Moodle:
www.decom.ufop.br/moodle
- Lista de e-mails:
bcc221-decom@googlegroups.com
- Para solicitar acesso:
<http://groups.google.com/group/bcc221-decom>



UFOP



Avisos



Avisos

UFOP



Na aula passada

UFOP

- Tratamento de Exceções
- *try e catch*
- Modelo de Terminação
- Cláusula *throws*
- Quando Utilizar Exceções?
- Hierarquia de Exceções Java
- Blocos *finally*
- *throw*
- Desfazendo a Pilha
- *printStackTrace, getStackTrace e getMessage*
- Exceções Encadeadas
- Declarando Novos Tipos de Exceções
- Pré-Condições e Pós-Condições
- Asserções



Na aula de hoje

UFOP

- Genéricos
 - Métodos Genéricos
 - Classes Genéricas
 - Tipos "Crus"
 - Coringas em Métodos Genéricos
 - Genéricos e Herança
- Coleções
 - Classe *Arrays*
 - Interface *Collection* e Classe *Collections*
 - Listas
 - *ArrayList* e *Iterator*
 - *LinkedList*
 - *Vector*
 - Algoritmos
 - Pilhas
 - Filas de Prioridade
 - Conjuntos
 - Mapas



Genéricos

UFOP

- 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;
 - Também fornece **segurança de tipo**
 - Permite aos programadores detectar tipos inválidos em tempo de compilação.

Métodos Genéricos



Métodos Genéricos

UFOP

- 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

UFOP

- 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.
- 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

UFOP

- 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.



GenericMethodTest.java

```
public class GenericMethodTest {  
    //metodo generico  
    public static < E > void printArray( E[] inputArray ){  
        // exibe 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, 7.7 };  
        Character[] characterArray = { 'H', 'E', 'L', 'L', 'O' };  
  
        System.out.println( "Array integerArray contains:" );  
        printArray( integerArray ); // envia um vetor de Integer  
        System.out.println( "\nArray doubleArray contains:" );  
        printArray( doubleArray ); // envia um vetor de Double  
        System.out.println( "\nArray characterArray contains:" );  
        printArray( characterArray ); // envia um vetor de Character  
    }  
}
```



Métodos Genéricos

UFOP

- 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

UFOP

- 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

UFOP

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

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



Métodos Genéricos

UFOP

- 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



Classes Genéricas

UFOP

- 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

UFOP

- 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

UFOP

- **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;
- Em Java há 8 classes empacotadoras
 - *Byte, Short, Integer, Long, Float, Double, Character* e *Boolean*;
 - Embora não seja considerada uma classe empacotadora, a classe *Void* é similar
 - No sentido de fornecer uma representação de classe para o tipo *void*.
 - Todas declaradas no pacote `java.lang`.



Classes Genéricas

UFOP

- 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.



Stack.java

UFOP

```
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

UFOP

```
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

UFOP

- 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.
- 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

UFOP

- 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

UFOP

```
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 );

        testPushDouble();
        testPopDouble();
        testPushInteger();
        testPopInteger();
    }
}
```



StackTest2.java

UFOP

```
public class StackTest2
{
    private Double[] doubleElements = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6 };
    private Integer[] integerElements = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };

    private Stack< Double > doubleStack; // pilha de objetos Double
    private Stack< Integer > integerStack; // pilha de objetos Integer

    // testa os objetos da pilha
    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 );
    }
}
```



StackTest2.java

UFOP

```
// 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();
    }
}
```



StackTest2.java

```
// 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();
    }
}

public static void main( String args[] )
{
    StackTest2 application = new StackTest2();
    application.testStacks();
}
```

Tipos “Crus”



Tipos “Crus”

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

UFOP

- Os tipos crus são importante para a compatibilidade das versões antigas do Java
  - Por exemplo, as estruturas do ***Java Collections Framework*** armazenavam referências à classe *Object* e agora são classes genéricas;
  - 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



# Coringas em Métodos Genéricos

UFOP

- 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

UFOP

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

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

 return total;
}
```



# Coringas em Métodos Genéricos

UFOP

- 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 e Herança

UFOP

- 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



# Coleções

UFOP

- O *Java Collections Framework* contém estruturas de dados “pré-empacotadas”, interfaces e algoritmos para manipular tais estruturas
  - Relação muito próxima com genéricos;
  - Podemos utilizar as estruturas sem nos preocuparmos com detalhes da implementação interna;
  - Excelente desempenho com bons tempos de execução e minimização do uso de memória;
  - Também fornece **iteradores**, para percorrermos os elementos de uma coleção;
  - Reuso de *software*.

<http://docs.oracle.com/javase/8/docs/api/java/util/Collections.html>



# Coleções

UFOP

- Uma coleção é um objeto que mantém referências a outros objetos
  - Normalmente, objetos de um mesmo tipo.
- 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

UFOP

| 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**



# Classe *Arrays*

UFOP

- 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

UFOP

```
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

UFOP

```
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: ");
 for (int intValue : filledIntArray)
 System.out.printf("%d ", intValue);

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

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



# UsingArrays.java

UFOP

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

// compara o conteúdo 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

UFOP

```
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");
}
```



# Saída

UFOP

```
doubleArray: 0.2 3.4 79 8.4 9.3
intArray: 1 2 3 4 5 6
filledIntArray: 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*



# Interface *Collection* e Classe *Collections*

UFOP

- A interface ***Collection*** é a classe base da hierarquia de todas interfaces de coleções
  - Contém operações realizadas em coleções inteiras (***bulk operations***)
    - Adicionar elementos;
    - Esvaziar;
    - Comparar;
    - Reter elementos.
  - Também contém operações que retornam iteradores (objetos ***Iterator***), que nos permitem percorrer uma coleção.
- Para permitir o comportamento polimórfico, geralmente os parâmetros de métodos são do tipo *Collection*
  - Além disto, uma coleção pode ser convertida em um vetor.

# Interface *Collection* e Classe *Collections*



UFOP

- 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



# Listas

UFOP

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

# *ArrayList e Iterator*



# *ArrayList e Iterator*

UFOP

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



# CollectionTest.java

UFOP

```
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);

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



# CollectionTest.java

UFOP

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

// exibe 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: ");

// exibe o conteudo
for (String color : list)
 System.out.printf("%s ", color);
}
```



# CollectionTest.java

```
// 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();
}
```



# Saída

UFOP

ArrayList:

MAGENTA RED WHITE BLUE CYAN

ArrayList after calling removeColors:

MAGENTA CYAN



# ArrayList e Iterator

UFOP

- 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.
- O método **contains** determina a pertinência de um elemento em uma coleção;
- 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**.

# *LinkedList*



# LinkedList

UFOP

- 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.



# ListTest.java

UFOP

```
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);

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



# ListTest.java

UFOP

```
list1.addAll(list2); // concatena as listas
list2 = null; // libera
printList(list1); // exibe os elementos

convertToUppercaseStrings(list1); // converte para maiusculas
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
printReversedList(list1); // exibe os elementos na ordem inversa
}

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

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

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



# ListTest.java

UFOP

```
// 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
 }
}

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



# ListTest.java

UFOP

```
// 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();
}
```

# Saída



UFOP

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

UFOP

- 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;
- 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

UFOP

- 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

UFOP

```
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

UFOP

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

System.out.println("colors: ");

for (String color : colors)
 System.out.println(color);
}

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



# Saída

UFOP

colors:

cyan

black

blue

yellow

green

red

pink



# LinkedList

UFOP

- Uma vez que obtemos uma *List* criada pelo método *asList*, 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.
- A partir de uma *List* também podemos obter um vetor com os mesmos elementos
  - Método ***toArray***.

# *Vector*



# Vector

UFOP

- Assim como *ArrayLists*, os ***Vectors*** fornecem uma estrutura parecida com um vetor, que 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

UFOP

```
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);

 // 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());
 }
 }
}
```



# VectorTest.java

UFOP

```
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", vector.indexOf("red"));
else
 System.out.println("\n\"red\" not found\n");

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");
```



# VectorTest.java

```
System.out.printf("\nSize: %d\nCapacity: %d\n", vector.size(), vector.capacity());
}

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();
}
}
```



# Saída

UFOP

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



# Algoritmos

UFOP

- 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
    - Ou seja, podem ser aplicados a objetos de classes que implementam interfaces específicas, independente de detalhes internos.
- 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

UFOP

| 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



# Pilhas

UFOP

- 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*.



# StackTest.java

UFOP

```
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;

 // usa o metodo push
 stack.push(longNumber);
 printStack(stack);
 stack.push(intNumber);
 printStack(stack);
 stack.push(floatNumber);
 printStack(stack);
 stack.push(doubleNumber);
 printStack(stack);
 }
}
```



# StackTest.java

UFOP

```
// remove os itens da pilha
try
{
 Number removedObject = null;

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



# StackTest.java

UFOP

```
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 através 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();
}
```



# Saída

UFOP

```
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)
```

# Filas de Prioridade



# Filas de Prioridade

UFOP

- 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.
- 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

UFOP

```
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);

 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
 }
 }
}
```



# Saída

UFOP

Polling from queue: 3.2 5.4 9.8

# Conjuntos



# Conjuntos

UFOP

- 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.
- 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

UFOP

```
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

UFOP

```
//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();
}
```



# Saída

UFOP

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



# Mapas

UFOP

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

UFOP

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



# WordTypeCount.java

UFOP

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

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

 // 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

UFOP

```
private void displayMap()
{
 Set< String > keys = map.keySet(); // obtém 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());
}

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



# Saída

UFOP

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

UFOP

- 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*.
- 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.



UFOP



# Perguntas?



# Na próxima aula

- Arquivos



UFOP



**FIM**