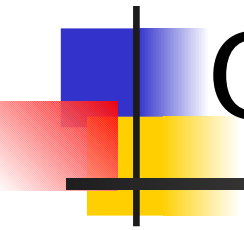


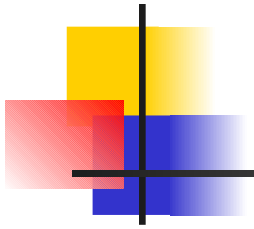
Computação Orientada a Objetos

Prof. Marcos L. Chaim

Genéricos

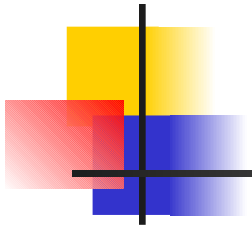
Slides elaborados pela Profa. Patrícia R. Oliveira

Classes empacotadoras de tipo para tipos primitivos



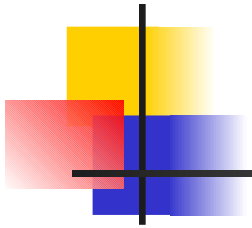
- Todas as classes em Java herdam, direta ou indiretamente, da classe **`Object`** (pacote **`java.lang`**)
- Classes que implementam estruturas de dados desenvolvidas em Java manipulam e compartilham objetos da classe **`Object`**
- Essas classes não podem manipular variáveis de tipos primitivos, mas podem manipular objetos de classes empacotadoras de tipos

Classes empacotadoras de tipo para tipos primitivos



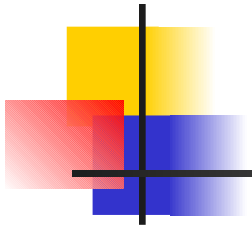
- Todo tipo primitivo tem uma classe empacotadora de tipo correspondente (no pacote **`java.lang`**)
  - **`Boolean`**
  - **`Byte`**
  - **`Character`**
  - **`Double`**
  - **`Float`**
  - **`Integer`**
  - **`Long`**
  - **`Short`**

Classes empacotadoras de tipo para tipos primitivos



- Toda classe empacotadora de tipo permite manipular valores de tipo primitivo como objetos
- Obs: classes empacotadoras de tipo são **`final`**, então não é possível estendê-las

Classes empacotadoras de tipo para tipos primitivos



- Métodos relacionados a um tipo primitivo encontram-se na classe empacotadora de tipo correspondente

- Ex: O método **`parseInt`**, que converte uma **`String`** em um valor **`int`**, encontra-se na classe **`Integer`**

Autoboxing e auto-unboxing

- Uma conversão boxing converte um valor de um tipo primitivo em um objeto da classe empacotadora de tipo correspondente
- Uma conversão unboxing converte um objeto de uma classe empacotadora de tipo em um valor do tipo primitivo correspondente
- J2SE 5.0 permite que essas conversões sejam feitas automaticamente (chamadas de autoboxing e auto-unboxing)

Autoboxing e auto-unboxing

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


//autoboxing
integerArray[0] = 10; // atribui Integer 10 a
   integerArray[0]


//auto-unboxing
int value = integerArray[0]; // obtem valor int de
   Integer
```

Introdução aos genéricos

- Questão 1:

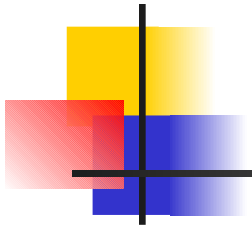
O que fazer para escrever um único método de ordenação **`ordena`** para elementos em um array de **`Integer`**, em um array de **`String`** ou em um array de qualquer tipo que suporte ordenamento?

Introdução aos genéricos

- Questão 2:

O que fazer para escrever uma única classe **`Stack`** que seria utilizada como uma pilha de inteiros, uma pilha de números de ponto flutuante, uma pilha de **`String`** ou uma pilha de qualquer outro tipo?

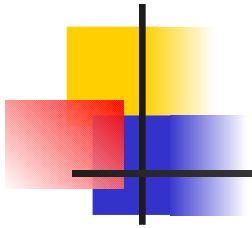
Introdução aos genéricos



- Questão 3:

O que fazer para detectar não-correspondências de tipos em tempo de compilação (segurança de tipos em tempo de compilação)?

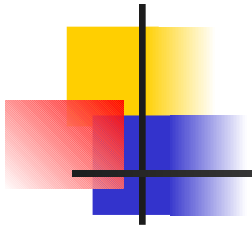
Ex: se uma pilha armazenasse somente inteiros, tentar inserir uma **`String`** nessa pilha.



Introdução aos genéricos

- Genéricos: recurso que fornece um meio de criar os objetos gerais citados nas Questões 1, 2 e 3

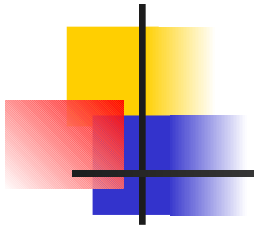
- Classes genéricas: permite que o programador defina, com uma única declaração de classe, um conjunto de tipos relacionados



Introdução aos genéricos

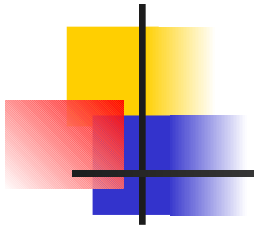
- Métodos genéricos: permite que o programador defina, com uma única declaração de método, um conjunto de métodos relacionados
- Os genéricos também fornecem segurança de tipo em tempo de compilação, permitindo a detecção de tipos inválidos em tempo de compilação

Exemplos de aplicações com Genéricos



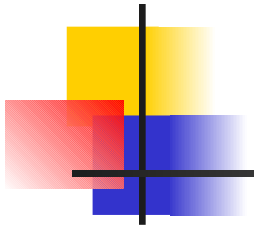
- 1) Escrever um método genérico para ordenar um objeto array e então invocar esse mesmo método com arrays de **`Integer`**, arrays de **`Double`**, arrays de **`String`** para ordenar os elementos no array.

Exemplos de aplicações com Genéricos



- 2) Permitir ao compilador realizar uma verificação de tipo para assegurar que o array passado para o método de ordenação contenha elementos do mesmo tipo.

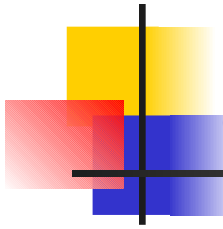
Exemplos de aplicações com Genéricos



- 3) Escrever uma única classe **`Stack`** genérica que manipulasse uma pilha de objetos e instanciasse objetos **`Stack`** em uma pilha de **`Integer`**, uma pilha de **`Double`**, uma pilha de **`String`**, etc.

- Obs: O compilador realizaria a verificação de tipo para assegurar que a estrutura **`Stack`** armazena elementos do mesmo tipo.

Motivação para métodos genéricos



- Métodos sobrecarregados são bastante utilizados para realizar operações semelhantes em tipos diferentes de dados
- Ex: Utilização de três métodos **`printArray`** sobrecarregados para imprimir um array de tipos diferentes.

Ex: métodos de impressão para arrays de tipos diferentes

- Método **OverloadedMethods.printArray** para imprimir um array de **Integer**

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

   System.out.println();
} // fim do método printArray
```

Ex: métodos de impressão para arrays de tipos diferentes

- Método **OverloadedMethods.printArray** para imprimir um array de **Double**

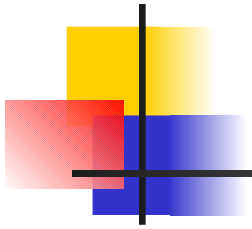
```
public static void printArray( Double[] inputArray )
{
   // exibe elementos do array
   for ( Double element : inputArray )
      System.out.printf( "%s ", element );

   System.out.println();
} // fim do método printArray
```

Ex: métodos de impressão para arrays de tipos diferentes

- Método **OverloadedMethods.printArray** para imprimir um array de **Character**

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

   System.out.println();
} // fim do método printArray
```

Ex: métodos de impressão para arrays de tipos diferentes



- Quando o compilador encontra uma chamada de método, ele sempre tenta localizar uma declaração de método com o mesmo nome de método e parâmetros que correspondam aos tipos de argumento da chamada
- No exemplo, cada chamada a printArray corresponde exatamente a uma das declarações desse método

Ex: métodos de impressão para arrays de tipos diferentes

```
...
printArray( integerArray );
...
```

O compilador determina o tipo do argumento **`integerArray`** (ie, **`Integer[]`**) e tenta localizar um método chamado **`printArray`** que especifica um único parâmetro **`Integer[]`**

Ex: métodos de impressão para arrays de tipos diferentes

- Método **`OverloadedMethods.main`**

```
public static void main( String args[] )
{
   // cria arrays de 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 contem:" );
   printArray( integerArray ); // passa um array de Integers
   System.out.println( "\nArray doubleArray contem:" );
   printArray( doubleArray ); // passa um array Doubles
   System.out.println( "\nArray characterArray contem:" );
   printArray( characterArray ); // passa um array de Characters
} // fim de main
```

Ex: métodos de impressão para arrays de tipos diferentes

- Saída do programa:

```
Array integerArray contem:
1 2 3 4 5 6

Array doubleArray contem:
1.1 2.2 3.3 4.4 5.5 6.6 7.7

Array characterArray contem:
H E L L O
```

Ex: métodos de impressão para arrays de tipos diferentes

- Nesse exemplo, os tipos de elementos dos arrays aparecem em:
  - Nos cabeçalhos dos métodos

```
public static void printArray( Integer[] inputArray )
public static void printArray( Double[] inputArray )
public static void printArray( Character[] inputArray )
```

  - Nas instruções for

```
for ( Integer element : inputArray )
for ( Double element : inputArray )
for ( Character element : inputArray )
```

Ex: métodos de impressão para arrays de tipos diferentes

- Se os tipos dos elementos em cada método fossem substituídos por um nome de tipo genérico **E**, então os três métodos de impressão seriam iguais a:

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

   System.out.println();
} // fim do método printArray
```

Ex: métodos de impressão para arrays de tipos diferentes

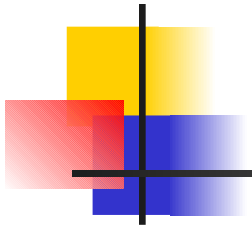
- Utilizando um tipo genérico, é possível declarar um método **`printArray`** que pode exibir as representações string dos elementos de qualquer array que contém objetos

```
public static <E> void printArray( E[] inputArray )
{
   // exibe elementos do array
   for ( E element : inputArray )
      System.out.printf( "%s ", element );

   System.out.println();
} // fim do método printArray
```

Ex: métodos de impressão para arrays de tipos diferentes

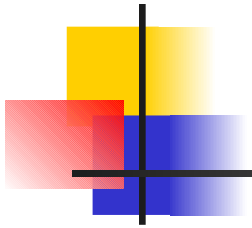
- O especificador de formato **%s** poder utilizado para gerar saída de qualquer objeto de representação de string – método **`toString`** é chamado implicitamente

```
public static <E> void printArray( E[] inputArray )
{
   // exibe elementos do array
   for ( E element : inputArray )
      System.out.printf( "%s ", element );

   System.out.println();
} // fim do método printArray
```



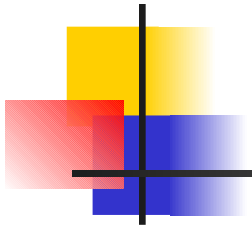
Métodos genéricos

- Se as operações realizadas por métodos sobrecarregados forem idênticas para cada tipo de argumento, esses métodos podem ser codificados por métodos genéricos.
  - Representação mais compacta e conveniente
- Pode-se escrever uma única declaração de método genérico que pode ser chamada com argumentos de tipos diferentes.



Métodos genéricos

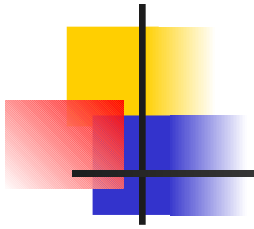
- Tradução em tempo de compilação:
  - Com base nos tipos dos argumentos passados para o método genérico, o compilador trata cada chamada do método de forma apropriada.



Métodos genéricos - Exemplo

- Implementando o método **`printArray`** genérico, as chamadas a esse método e as saídas do programa permanecem as mesmas.
  - demonstra o poder expressivo dos genéricos

Declaração de métodos genéricos

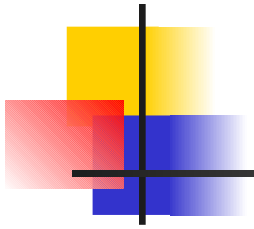


- Todas as declarações de métodos genéricos têm uma seção de parâmetros de tipos, delimitada por colchetes angulares (ex, **<E>**) que precedem o tipo de retorno do método
- Cada seção de parâmetros de tipos contém um ou mais parâmetros de tipos, separados por vírgulas
- Um parâmetro de tipo (também conhecido como variável de tipo) é um identificador que especifica um nome genérico do tipo

Declaração de métodos genéricos

- Os parâmetros de tipo na declaração de um método genérico podem ser utilizados para especificar:
  - o tipo de retorno
  - tipos de parâmetros
  - tipos de variáveis locais
- Parâmetros de tipo atuam também como marcadores de lugar para os tipos dos argumentos passados ao método genérico, conhecidos como argumentos de tipos reais

Declaração de métodos genéricos



- O corpo de um método genérico é declarado como o de qualquer outro método
- Os parâmetros de tipo podem representar somente tipos por referência – não tipos primitivos, como **`int`**, **`double`** e **`char`**
- Os nomes dos parâmetros de tipo por toda a declaração do método devem corresponder àqueles declarados na seção de parâmetro de tipo

Declaração de métodos genéricos - Exemplo

- Na instrução **`for, element`** é declarado como tipo **E**, que corresponde ao parâmetro de tipo (**E**), declarado no cabeçalho do método

```
public static <E> void printArray( E[] inputArray )
{
   // exibe elementos do array
   for ( E element : inputArray )
      System.out.printf( "%s ", element );

   System.out.println();
} // fim do método printArray
```

Declaração de métodos genéricos

- Um parâmetro de tipo pode ser declarado somente uma vez na seção de parâmetro de tipo, mas pode aparecer mais de uma vez na lista de parâmetros do método
- Ex:

```
public static <E> void printTwoArrays(E[] array1, E[] array2)
```

- Os nomes de parâmetros de tipo não precisam ser únicos entre diferentes métodos genéricos

Declaração de métodos genéricos - Exemplo

- A seção de parâmetro de tipo do método **printArray,** declara o parâmetro de tipo **E** como o marcador de lugar para o tipo de elemento do array que o método enviará para a saída

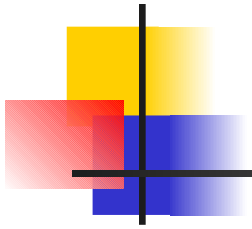
```
public static <E> void printArray( E[] inputArray )
{
   // exibe elementos do array
   for ( E element : inputArray )
      System.out.printf( "%s ", element );

   System.out.println();
} // fim do método printArray
```

Declaração de métodos genéricos - Exemplo

- A instrução **for** também utiliza **E** como o tipo de elemento

```
public static <E> void printArray( E[]
inputArray )
 {
    // exibe elementos do array
    for ( E element : inputArray )
       System.out.printf( "%s ", element );

    System.out.println();
 } // fim do método printArray
```

Declaração de métodos genéricos



- É recomendável que parâmetros de tipo sejam especificados como letras maiúsculas individuais
- Em geral, um parâmetro de tipo que representa o tipo de um elemento em um array (ou em outra estrutura de dados) é nomeado **E**, que representa “elemento”

Métodos genéricos – Tradução em tempo de compilação

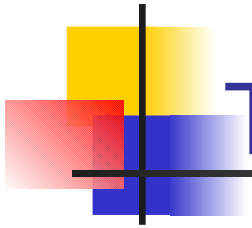
- Quando o compilador encontra a chamada **`printArray( integerArray)`**, ele primeiro determina o tipo do argumento **`integerArray`** (ie, **`Integer[]`**) e tenta encontrar um método **`printArray`** com um único parâmetro desse tipo. Não há tal método nesse exemplo!

```
public static void main( String args[] )
{
    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 contem:" );
   printArray( integerArray ); // passa um array de Integers
} // fim de main
```

Métodos genéricos – Tradução em tempo de compilação

- Em seguida, o compilador verifica que há um método genérico **`printArray`** que especifica um parâmetro de array individual, e utiliza o parâmetro de tipo para representar o tipo de elemento do array

```
public static void main( String args[] )
{
    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 contem:" );
   printArray( integerArray ); // passa um array de Integers
} // fim de main
```



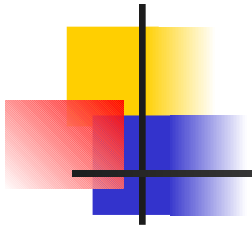
Métodos genéricos – Tradução em tempo de compilação

- O compilador também determina se as operações no corpo do método genérico podem ser aplicadas a elementos do tipo armazenado no argumento do array
- Quando o compilador traduz o método genérico em *bytecode* Java, ele remove a seção de parâmetros de tipo e substitui os parâmetros de tipo por tipos reais
- Esse processo é chamado de erasure

Métodos genéricos – Tradução em tempo de compilação

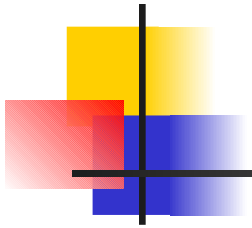
- Por padrão, todos os tipos genéricos são substituídos pelo tipo **`Object`**

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

    System.out.println();
 } // fim do método printArray
```



Classes genéricas

- O conceito de uma estrutura de dados, como uma pilha, pode ser entendido independentemente do tipo que ela manipula
- Classes genéricas fornecem um meio de descrever o conceito de uma pilha (ou de qualquer outra classe) de uma maneira independente do tipo
- É possível, então, instanciar objetos específicos de uma classe genérica

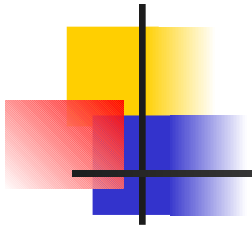


Classes genéricas

- Em tempo de compilação, o compilador Java
  - garante a segurança de tipo do seu código
  - utiliza técnicas de erasure para permitir que o código do seu cliente interaja com a classe genérica

Classes genéricas - Exemplo

- Uma classe **`Stack`** genérica poderia ser a base para criar várias classes **`Stack`**, por exemplo:
  - **`Stack`** de **`Double`**
  - **`Stack`** de **`Integer`**
  - **`Stack`** de **`Character`**
- Essas classes são conhecidas como classes parametrizadas ou tipos parametrizados porque aceitam um ou mais parâmetros



Classes genéricas

- Lembre-se que parâmetros de tipo só representam tipos por referência
  - a classe genérica **`Stack`** não pode instanciada com tipos primitivos
- Entretanto, é possível instanciar uma **`Stack`** que armazena objetos das classes empacotadoras Java e permitir que Java utilize o *autoboxing* para converter os valores primitivos em objetos

Declaração de classes genéricas

- A declaração de uma classe genérica se parece com a declaração de uma não-genérica, exceto que o nome da classe é seguido por uma seção de parâmetros de tipo

```
public class Stack< E >
{
   private final int size; // número de elementos na pilha
   private int top; // localização do elemento superior
   private E[] elements; // array que armazena elementos na pilha
//...
```

Declaração de classes genéricas

- O parâmetro de tipo **E** representa o tipo de elemento que a **Stack** manipulará
- O parâmetro de tipo **E** é utilizado por toda declaração da classe para representar o tipo do elemento

```
public class Stack< E >
{
   private final int size; // número de elementos na pilha
   private int top; // localização do elemento superior
   private E[] elements; // array que armazena elementos na pilha
//...
```

Declaração de classes genéricas

- A classe **`Stack`** declara a variável **`elements`** como um array do tipo **`E`**
- Esse array armazenará os elementos da **`Stack`**
- Como criar esse array?

```
public class Stack< E >
{
   private final int size; // número de elementos na pilha
   private int top; // localização do elemento superior
   private E[] elements; // array que armazena elementos na pilha
//...
```

Declaração de classes genéricas

- Não é permito usar parâmetros de tipo em expressões de criação de arrays porque este parâmetro (no caso, **E**) não estará disponível em tempo de execução
- Solução: criar um array do tipo **Object** e fazer uma coerção na referência retornada por new para o tipo **E[]**

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

Declaração de classes genéricas - Exemplo

- Método **push.Stack**

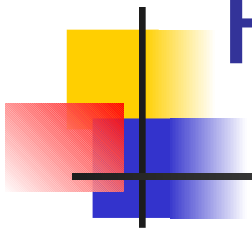
```
// insere o elemento na pilha; se bem-sucedido retorna true;
// caso contrário, lança uma FullStackException
public void push( E pushValue )
{
   if ( top == size - 1 ) // se a pilha estiver cheia
      throw new FullStackException( String.format(
         "Stack is full, cannot push %s", pushValue ) );

   elements[ ++top ] = pushValue; // insere pushValue na Stack
} // fim do método push
```

Declaração de classes genéricas - Exemplo

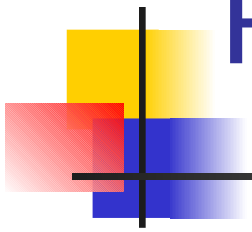
- Método **pop** . **Stack**

```
// retorna o elemento superior se não estiver vazia; do
  contrário lança uma EmptyStackException
 public E pop()
 {
    if ( top == -1 ) // se pilha estiver vazia
       throw new EmptyStackException( "Stack is empty,
  cannot pop" );

    return elements[ top-- ]; // remove e retorna o
  elemento superior da Stack
 } // fim do método pop
```

Resumo



- Foram apresentados os mecanismos para criação de métodos e classes genéricos.
- Nestes métodos e classes é, especifica-do um tipo genérico <E> que pode ser instanciado para diferentes classes.
- Tipos primitivos (int, boolean, double) não podem ser instanciados. Porém, suas classes empacotadoras, sim.



Referências

- H.M. Deitel & P. J. Deitel, Java – como programar, 6a. Edição, Pearson Prentice-Hall, São Paulo, 2005. Capítulo 18.