Generics - Java

Fernando Santos

Programação Orientada a Objetos

Generics

- Generics é uma funcionalidade incorporada ao Java a partir da versão 5.0
- Permite aos programadores escreverem **métodos genéricos**
  - Os parâmetros dos métodos, variáveis locais e o tipo de retorno podem ser definidos na chamada do método
  - Permite ao mesmo método ser invocado usando-se tipos distintos (sem precisar sobrescrevê-lo)
- Permite também a definição de **classes genéricas**
  - Os atributos da classe podem ser definidos no momento da instanciação do objeto
  - Recurso útil ao definir classes como estruturas de dados
- **Generics** em Java oferece os mesmos recursos dos **Templates** em C++

Generics

- Considere o método *imprimeVetor* da classe abaixo

```
class Teste
{
    public static void imprimeVetor(double v[]){
        for ( double e : v ) System.out.printf(e + " ");
        System.out.println();
    }

    public static void main(String args[])
    {
        double[] arrayDouble = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6};
        System.out.println("Vetor de double: ");
        imprimeVetor(arrayDouble);

        int[] arrayInt = {1, 2, 3, 4, 5, 6};
        System.out.println("Vetor de inteiros: ");
        imprimeVetor(arrayInt);
    }
}
```

- Ao tentar compilar o código um erro é apresentado
- Embora o tipo **double** contenha o tipo **int**, uma referência para **double** nao pode referenciar um vetor de **int**
- Seria necessário ter 2 implementações de *imprimeVetor*

Generics

- O problema se agravaria, à medida em que fosse necessário imprimir vetores de outros tipos de dados
  - Para cada tipo, uma nova implementação
  - Cada implementação teria apenas o cabeçalho do método (e o tipo usado no **for**) diferente das demais

```
class Teste
{
    public static void imprimeVetor(double v[]){
        for ( double e : v ) System.out.printf(e + " ");
        System.out.println();
    }

    public static void main(String args[])
    {
        double[] arrayDouble = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6};
        System.out.println("Vetor de double: ");
        imprimeVetor(arrayDouble);

        int[] arrayInt = {1, 2, 3, 4, 5, 6};
        System.out.println("Vetor de inteiros: ");
        imprimeVetor(arrayInt);

        char[] arrayChar = {'E', 'C', 'O', '0', '3', '0'};
        System.out.println("Vetor de char: ");
        imprimeVetor(arrayChar);
    }
}
```

Generics

- Este problema seria facilmente resolvido ao implementar um método genérico

```
class Teste
{
    public static < T > void imprimeVetor(T v[]){
        for ( T e : v ) System.out.printf(e + " ");
        System.out.println();
    }

    public static void main(String args[])
    {
        Double[] arrayDouble = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6};
        System.out.println("Vetor de double: ");
        imprimeVetor(arrayDouble);

        Integer[] arrayInt = {1, 2, 3, 4, 5, 6};
        System.out.println("Vetor de inteiros: ");
        imprimeVetor(arrayInt);

        Character[] arrayChar = {'E', 'C', 'O', '0', '3', '0'};
        System.out.println("Vetor de char: ");
        imprimeVetor(arrayChar);
    }
}
```

- Define-se um ou mais tipos genéricos, que serão delimitados pelos símbolos $< >$
- Estes tipos são definidos na chamada do método

Generics

- Uma ressalva: métodos genéricos (e classes genéricas) podem ser definidos apenas para tipos referenciáveis
- Logo, não podem ser definidos para tipos primitivos
  - {**byte**, **short**, **int**, **long**, **float**, **double**, **boolean**, **char**}
- Essa limitação é contornada usando-se as **classes empacotadoras de tipo**, que são uma alternativa oferecida por Java para tratar tipos primitivos como referenciáveis de forma transparente
  - {**Byte**, **Short**, **Integer**, **Long**, **Float**, **Double**, **Boolean**, **Character**}

Generics

- Mas qual seria a diferença entre usar um método genérico e um método usando **Object**, como abaixo?

```
class Teste
{
    public static void imprimeVetor(Object v[]){
        for ( Object e : v ) System.out.printf(e + " ");
        System.out.println();
    }

    public static void main(String args[])
    {
        Double[] arrayDouble = {0.0, 6.6, 9.9, 4.4, 5.5, 2.2};
        System.out.println("Vetor de double: ");
        imprimeVetor(arrayDouble);

        Integer[] arrayInt = {10, 7, 13, 4, 9, 6};
        System.out.println("Vetor de inteiro: ");
        imprimeVetor(arrayInt);

        Character[] arrayChar = {'E', 'C', 'O', '0', '3', '0'};
        System.out.println("Vetor de char: ");
        imprimeVetor(arrayChar);
    }
}
```

Generics

- Mas qual seria a diferença entre usar um método genérico e um método usando **Object**, como abaixo?

```
class Teste
{
    public static void imprimeVetor(Object v[]){
        for ( Object e : v ) System.out.printf(e + " ");
        System.out.println();
    }

    public static void main(String args[])
    {
        Double[] arrayDouble = {0.0, 6.6, 9.9, 4.4, 5.5, 2.2};
        System.out.println("Vetor de double: ");
        imprimeVetor(arrayDouble);

        Integer[] arrayInt = {10, 7, 13, 4, 9, 6};
        System.out.println("Vetor de inteiro: ");
        imprimeVetor(arrayInt);

        Character[] arrayChar = {'E', 'C', 'O', '0', '3', '0'};
        System.out.println("Vetor de char: ");
        imprimeVetor(arrayChar);
    }
}
```

- Para este caso em específico, a diferença é NENHUMA!

Generics

- Java implementa métodos e classes genéricas da seguinte forma
  - substitui os tipos genéricos por tipos que sejam de uma superclasse contenha todas as candidatas a serem usadas
  - realiza o casting de tipos (quando necessário) implicitamente
  - esse processo é conhecido como **erasure**
- No exemplo anterior, o tipo **Object** seria usado pois é o único tipo capaz de referenciar qualquer objeto que invoque o método

Obs.: A abordagem do **erasure** é diferente da utilizada nos **templates** em C++. Lá o compilador gera códigos para sobrescrever o método considerando cada tipo que o invoca

Generics

- Considere agora a função genérica *maior* abaixo

```
class Teste
{
    public static < T > void imprimeVetor(T v[]){
        for ( T e : v ) System.out.printf(e + " ");
        System.out.println();
    }

    public static < T extends Comparable< T > > T maior(T v[]){
        T max = v[0];
        for ( T e : v ) { if (e.compareTo(max) > 0 ) max = e; }
        return max;
    }

    public static void main(String args[])
    {
        Double[] arrayDouble = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6};
        System.out.println("Vetor de double: ");
        imprimeVetor(arrayDouble);
        System.out.println("Maior elemento: " + maior(arrayDouble));

        Integer[] arrayInt = {1, 2, 3, 4, 5, 6};
        System.out.println("Vetor de inteiros: ");
        imprimeVetor(arrayInt);
        System.out.println("Maior elemento: " + maior(arrayInt));

        Character[] arrayChar = {'E', 'C', 'O', '0', '3', '0'};
        System.out.println("Vetor de char: ");
        imprimeVetor(arrayChar);
        System.out.println("Maior elemento: " + maior(arrayChar));
    }
}
```

Generics

- Neste caso, substituir o tipo genérico por **Object** não forneceria uma implementação equivalente
- Isto porque o tipo genérico deve implementar a interface **Comparable**
  - Generics considera a palavra-chave **extends** tanto para denotar classes que herdam superclasses quando para classes que implementam uma interface
- Ao fazer isto, é possível usar o método *compareTo* do tipo genérico (o que não seria possível utilizando **Object**)
- Além disto, Generics realiza o *casting* implicito do tipo de retorno onde o método foi invocado

Generics

- O conceito de Generics pode ser estendido às classes
- Como exemplo, considere a implementação (não genérica) da classe Pilha abaixo

```
class Pilha
{
    private int max, topo;
    private int[] elementos;

    public Pilha(int max)
    {
        topo = -1;
        this.max = max;
        elementos = new int[max];
    }
    public void push(int e) throws Exception
    {
        if (topo < (max-1)) elementos[++topo] = e;
        else throw new Exception();
    }
    public int pop() throws Exception
    {
        if (topo >= 0) return elementos[topo--];
        else throw new Exception();
    }
}
```

Generics

- A classe Pilha foi implementada para armazenar apenas números inteiros
- Mesmo que a estrutura da Pilha seja a mesma, ela nao poderia armazenar dados de outros tipos
- Para resolver este problema, é possível definir a classe Pilha genericamente, de modo que ela seja capaz de armazenar qualquer tipo de dado (desde que seja não-primitivo)
- Generics auxilia na implementação de estruturas de dados
    - Diversas classes que compõe o **Collections** Java são definidas de forma genérica

Generics

- A implementação abaixo define uma Pilha genérica

```
class Pilha < T >
{
    private int max, topo;
    private T[] elementos;

    public Pilha(int max)
    {
        topo = -1;
        this.max = max;
        elementos = ( T[] ) new Object[max];
    }
    public void push(T e) throws Error
    {
        if (topo < (max-1)) elementos[++topo] = e;
        else throw new Error();
    }
    public T pop() throws Error
    {
        if (topo >= 0) return elementos[topo--];
        else throw new Error();
    }
}
```

- O tipo T pode ser usado na implementação da classe, mas será definido somente no momento da sua instanciação

Generics

- No código acima, são instanciadas 2 Pilhas: uma armazena objetos **Par**, a outra armazena Inteiros
- Para cada instanciação, T assume um tipo distinto na pilha
- Note que, o tipo de retorno de p1.pop() é **Par** e p2.pop() retorna um **Integer** (casting implícito)

```
class Par
{
    private int a, b;
    public Par(int a, int b){ this.a = a; this.b = b; }
    public String toString(){ return "(" + a + ", " + b + ")"; }
}

class Main
{
    public static void main(String args[])
    {
        Pilha<Par> p1 = new Pilha<Par>(10);
        p1.push(new Par(0,10)); p1.push(new Par(2,1)); p1.push(new Par(3,4));

        System.out.println(p1.pop() + "\n" + p1.pop() + "\n" + p1.pop());

        Pilha<Integer> p2 = new Pilha<Integer>(10);
        p2.push(-10); p2.push(100); p2.push(15);
        System.out.println(p2.pop() + "\n" + p2.pop() + "\n" + p2.pop());
    }
}
```