GENÉRICOS EM JAVA

Java Generics

Ana Laura KOKETSU
Faculdade Politécnica de Campinas

Peter JANDL JR
Faculdade de Jaguariúna
Faculdade Politécnica de Campinas

Resumo: Os tipos genéricos foi a mais sofisticada novidade da linguagem Java 5 e um dos maiores incentivos para migração para a nova versão. A utilização dos tipos genéricos veio para facilitar a vida do programador, pois está entre as capacidades mais poderosas para reutilização de software com segurança de tipo em tempo real. O tipo genérico possui um sistema estático de verificação de tipos, feito para encontrar erros como conversões ilegais ou chamadas de métodos inexistentes num objeto em tempo de compilação. Evitando que um trecho de código pouco visível venha causar comportamentos indesejáveis posteriormente.
Palavras-chave: Java; genéricos.

Abstract: The generic types was the most sophisticated newness of the Java 5 language and one of the biggest incentives for migration to the new version. The use of generic types came to facilitate the life of the programmer, it is among the most powerful capabilities for software reuse in safety-type in real time. The generic type have a static system of verification of types, made to find errors as illegal conversions or called of inexistent methods in a object compilation time. Avoiding a barely visible piece of code will cause undesirable behavior.
Keywords: Java; generics.

INTRODUÇÃO

Apesar da existência de muitas opiniões de que a utilização de tipos genéricos é desnecessária devido à facilidade de se trabalhar com *typecasting* (conversão explícita de tipos) na linguagem Java, os tipos genéricos possuem ótimos argumentos para sua adoção. Alguns destes argumentos serão apresentados e discutidos neste artigo, motivando a utilização dos genéricos.

Na linguagem Java tradicional todas as conversões implícitas funcionam perfeitamente, sem gerar problemas. Já as conversões explícitas, forçadas com *typecasts* são fáceis de trabalhar, mas podem gerar uma exceção

ClassCastException. O uso da linguagem com tipos genéricos é mais garantido, pois um programa que não possui *typecast* e que não gera nenhum aviso (*warning*) de compilação jamais irá gerar uma ClassCastException (Jandl, 2007).

O programador que escolheu o Java deve concordar com os tipos estáticos. Afinal, trata-se de uma das características fundamentais da tecnologia. Os tipos genéricos nada mais são que os tipos estáticos elevados às últimas conseqüências.

Os tipos genéricos, métodos genéricos e classes genéricas permitem que programadores especifiquem, com uma única declaração de método, um conjunto de métodos relacionados ou, com uma única declaração de classe, um conjunto de tipos relacionados. Os genéricos também fornecem segurança de tipo em tempo de compilação que permite aos programadores capturar tipos inválidos em tempo de compilação (JCA, 2008; Sun MicroSystems, 2008A).

Os tipos genéricos podem ser aplicados com vantagens na construção de métodos, de classes ou interfaces.

MÉTODOS GENÉRICOS E SOBRECARGA

A Listagem 1 mostra um programa que imprime elementos de um *array* de Integer, um *array* de Double e um *array* de Character. Para imprimir estes elementos sem a utilização dos tipos genéricos foi necessário criar três métodos distintos que realizam operações semelhantes em tipos diferentes de dados, gerando assim métodos sobrecarregados na classe denominada OverloadedMethods.

Listagem 1. Uso de métodos sobrecarregados para imprimir *arrays* de diferentes tipos.

```
public class OverloadedMethods {

   // método 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
```

```
   // método 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

   // método 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

   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 contém:" );
      printArray( integerArray );
      System.out.println( "Array doubleArray contém:" );
      printArray( doubleArray );
      System.out.println( "Array characterArray contém:" );
      printArray( characterArray );
   } // fim de main
} // fim da classe OverloadedMethods
```

Quando executado, o programa da Listagem 1 produz o seguinte resultado:

```
Array integerArray contém: 1 2 3 4 5 6
Array doubleArray contém: 1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array characterArray contém: H E L L O
```

O compilador sempre tenta localizar uma declaração de método com a mesma assinatura, ou seja, o nome de método indicado e parâmetros que correspondam aos tipos de argumentos na chamada de método.

Quando as operações realizadas por vários métodos sobrecarregados forem idênticas para cada tipo de argumento, os métodos podem ser codificados de maneira mais compacta por meio de um único método genérico. Uma única declaração de método genérico deve ser criada para ser chamada com argumentos de tipos diferentes.

No exemplo acima não é necessário criar três métodos semelhantes e sim um único método genérico printArray() que será responsável por imprimir elementos de qualquer *array* que contém objetos.

Todas as declarações de métodos genéricos têm uma seção de parâmetro de tipo delimitada por colchetes angulares (< E >) que precedem o tipo de retorno do método. Cada seção de parâmetro de tipo contém um ou mais parâmetros de tipos, separados por vírgula (Jandl, 2007; JCA, 2008).

Declarando printArray() como um método genérico, eliminamos a necessidade dos métodos sobrecarregados, poupando linhas de código, e criando um método reutilizável visto na Listagem 2.

Listagem 2. Uso de métodos genéricos para imprimir *arrays* de diferentes tipos.

```
public class GenericMethod {
   // método genérico printArray
   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

   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 contém:" );
      printArray( integerArray );
      System.out.println( "Array doubleArray contém:" );
      printArray( doubleArray );
      System.out.println( "Array characterArray contém:" );
      printArray( characterArray );
   } // fim de main
} // fim da classe GenericMethod
```

O resultado do programa da Listagem 2 é como visto a seguir:

```
Array integerArray contém: 1 2 3 4 5 6
Array doubleArray contém: 1.1 2.2 3.3 4.4 5.5 6.6 7.7
Array characterArray contém: H E L L O
```

Os resultados das Listagens 1 e 2 são idênticos, o que evidencia o poder expressivo dos genéricos: os métodos genéricos permitem reduzir a

quantidade de código produzida, enquanto sua aplicação é idêntica aos métodos sobrecarregados tradicionais, o que constitui uma dupla vantagem.

MÉTODO GENÉRICO E LIMITE DE TIPO

A Listagem 3 mostra um programa em que os tipos genéricos são utilizados no tipo de retorno e na lista de parâmetros. Neste exemplo temos um método maximum() que retorna o maior dos seus três argumentos do mesmo tipo. O parâmetro de tipo T garante que a lista de parâmetros e o retorno sejam todos do mesmo tipo, não é definido por uma classe, mas pelo próprio método.

Quando o compilador encontra a chamada do método maximum() com os inteiros 3, 4 e 5, ele primeiro procura um método maximum() que recebe três argumentos do tipo int. Não encontrando tal método, o compilador procura um método genérico que possa ser utilizado e encontra o método genérico maximum().

O método genérico maximum() pode ser chamado para tipos Integer, Double e String. O valor de T é modificado conforme os tipos passados como parâmetros, à única restrição é exigir que T seja compatível com a interface Comparable (Sun MicroSystems, 2008B). Esta restrição é chamada de limite de tipo.

Só é possível comparar dois objetos da mesma classe se esta implementar a interface genérica Comparable< T >. Todas as classes empacotadoras de tipo para tipos primitivos implementam esta interface. Objetos do tipo Comparable< T > sempre possuem o método compareTo().

Listagem 3. Método genérico maximum() retorna o maior dos três objetos.

```
public class Maximum {
   // determina o maior dos três objetos Comparable
   public static < T extends Comparable< T > >
                  T maximum( T x, T y, T z) {
      T max = x; // supõe que x é inicialmente o maior
      if ( y.compareTo( max ) > 0 )
         max = y; // y é o maior até agora
      if ( z.compareTo( max ) > 0 )
         max = z; // z é o maior
      return max; // retorna o maior objeto
   } // fim do método maximum
```

```
   public void main( String args[] ) {
      System.out.printf( "Máximo de %d, %d e %d é %d\n\n",
         3, 4, 5,maximum( 3, 4, 5 ) );
      System.out.printf(
         "Máximo de %.1f, %.1f e %.1f é %.1f\n\n",
         6.6, 8.8, 7.7, maximum( 6.6, 8.8, 7.7 ) );
      System.out.printf( "Máximo de %s, %s e %s é %s\n",
         "pêra", "maça", "laranja",
         maximum( "pêra", "maça", "laranja" ) );
   } // fim de main
} // fim da classe Maximum
```

A execução do programa da Listagem 3 apresenta o resultado que segue:

```
Máximo de 3, 4 e 5 é 5
Máximo de 6.6, 8.8 e 7.7 é 8.8
Máximo de pêra, maça e laranja é pêra
```

Com isso é possível observar que o método maximum() foi capaz de comparar corretamente elementos de tipos diferentes, sem necessidade de qualquer operação de *typecasting*.

CLASSES GENÉRICAS

Fornecem um meio de descrever o conceito de uma pilha ou de qualquer outra classe de uma maneira independente do tipo. Podemos instanciar objetos específicos de tipo da classe genérica. Essa capacidade fornece uma excelente oportunidade de reutilização de software.

Uma classe Stack genérica pode ser a base para criar muitas classes do tipo pilha ("*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.

A Listagem 4 mostra a implementação de uma classe genérica. A declaração de uma classe genérica se parece com a declaração de uma classe comum não genérica, exceto pelo fato de que o nome da classe é seguido por uma seção de parâmetro de tipo. O parâmetro de tipo E representa o tipo do elemento que a classe Stack manipulará. Este parâmetro é utilizado por toda a declaração da classe Stack para representar o tipo de elemento.

Listagem 4. Programa com classe genérica Stack.

```
// Classe Stack genérica
public class Stack < E > {
   private final int size; // numero de elementos na pilha
   private int top; // localização do elemento superior
   private E[] elements; // array para elementos da pilha

   // construtor default cria pilha do tamanho-padrão
   public Stack() {
      this(10);
   } // fim do construtor default da classe Stack

   // construtor cria pilha com número indicado de elementos
   public Stack( int s ) {
      size = s > 0 ? s : 10; // configura o tamanho de Stack
      top = -1; // Stack inicialmente vazia
      elements = ( E[] ) new Object[ size ]; //cria o array
   } // fim do construtor de Stack

   // insere elemento na pilha; se OK, retorna true;
   // caso contrario, lança uma FullStackException
   public void push( E pushValue ) {
      if ( top == size -1) // se a pilha estiver cheia
         throw new FullStackException( String.format(
            "Stack cheia, push %s nao realizado", pushValue ) );
      elements[++top] = pushValue; // insere pushValue na Stack
   }// fim do método push
} // fim da classe Stack< E >

// Classe de Exceção FullStackException
public class FullStackException extends RuntimeException {
   public FullStackException(String msg) {
      super(msg);
   }
}
// Programa de teste da classe genérica Stack
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 };
   // pilhas de objetos Double e Integer
   private Stack< Double > doubleStack;
   private Stack< Integer > integerStack;

   // teste objetos Stack
   public void testStacks() {
      // Stack de Doubles
      doubleStack = new Stack< Double >(5);
      // Stack de Integers
      integerStack = new Stack< Integer >(10);

      testPushDouble();  // insere doubles em doubleStack
      testPushInteger(); // insere ints em intStack
   } // fim do método testeStacks

   // testa o método push com a pilha de Double
   public void testPushDouble(){
      // insere elementos na pilha
      try {
```

```
            System.out.println( "Coloca elementos em doubleStack");
            // insere elementos na Stack
            for ( double element : doubleElements ) {
               System.out.printf( "%.1f ", element );
               doubleStack.push( element ); // insere elemento
            } // fim do for
         } catch ( FullStackException fullStackException ) {
            System.err.println();
            fullStackException.printStackTrace();
         } // fim da captura de FullStackException
      } // fim do método testPushDouble

      // testa o método push com a pilha de Integer
      public void testPushInteger () {
         // insere elementos na pilha
         try {
            System.out.println( "Coloca elementos em intStack" );
            // insere elementos na Stack
            for ( int element : integerElements ) {
               System.out.printf( "%d ", element );
               integerStack.push( element ); // insere elemento
            } // fim do for
         } catch ( FullStackException fullStackException ) {
            System.err.println();
            fullStackException.printStackTrace();
         } // fim da captura de FullStackException
      } // fim do método testPushInteger

      // programa principal
      public static void main(String a[]) {
         StackTest st = new StackTest();
         st.testStacks();
      }
   } // fim da clase StackTest
```

A execução do programa da Listagem 4 produz resultados como seguem:

```
Coloca elementos em doubleStack
1,1 2,2 3,3 4,4 5,5 6,6
FullStackException: Stack cheia, push 6.6 nao realizado
        at Stack.push(Stack.java:22)
        at StackTest.testPushDouble(StackTest.java:30)
        at StackTest.testStacks(StackTest.java:18)
        at StackTest.main(StackTest.java:57)
Coloca elementos em intStack
1 2 3 4 5 6 7 8 9 10 11
FullStackException: Stack cheia, push 11 nao realizado
        at Stack.push(Stack.java:22)
        at StackTest.testPushInteger(StackTest.java:46)
        at StackTest.testStacks(StackTest.java:19)
        at StackTest.main(StackTest.java:57)
```

Com estes resultados é possível observar o uso de uma classe genérica. Também vale a pena destacar que na classe StackTest, os *arrays* criados são de tipos primitivos int e double, enquanto os tipos especificados para a pilha são das

classes wrapper Integer e Double. A movimentação de valores entre variáveis e as pilhas acontece sem operações de *typecasting* ou conversão devido as características de *autoboxing* e *autounboxing* adicionadas ao Java desde a versão 5 (Jandl, 2007).

ARRAY GENÉRICO

Não é possível criar um *array* genérico, pois o mecanismo genérico não permite parâmetros de tipo em expressões de criação de *arrays* porque o parâmetro de tipo não está disponível em tempo de execução. Só existe uma maneira de contornar este problema:

```
elements = ( E[] ) new object [ size ];
```

Esta linha de código é compilada, mas exige a criação de um *array* do tipo Object com *typecast* para o tipo [E]. Se a verificação estrita de tipos genéricos do compilador for acionada, o *typecast* irá gerar um *warning*. A razão deste *warning* é que o compilador não pode assegurar 100% de certeza que um *array* do tipo Object nunca conterá objetos de outros tipos além de E.

TIPOS BRUTOS

Na Listagem 4 a classe genérica instância Stacks com argumentos do tipo Double e Integer. Também é possível instanciar a classe genérica Stack sem especificar um argumento de tipo:

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

Nesse caso, diz-se que o objetoStack tem um tipo bruto, o que significa que o compilador utiliza implicitamente o tipo Object por toda a classe genérica para cada argumento de tipo.

Tipos brutos são uma facilidade que permite o uso de tipos genéricos como se não fossem genéricos, de forma a manter a compatibilidade com o código antigo não preparado para a exigência de parâmetros de tipo (retrocompatibilidade).

Considere as coleções que armazenam referências a Object, mas agora são implementados como tipos genéricos. Imagina se a linguagem exigisse que um tipo genérico fosse usado somente como tal, isso quebraria a compatibilidade com bilhões de linhas de códigos. Então a solução foi permitir estas declarações não parametrizadas.

Uma variável Stack de tipo bruto pode ser atribuída a uma Stack que especifica um argumento de tipo:

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

De maneira semelhante, uma variável Stack que especifica um argumento de tipo na sua declaração pode ser atribuída a um tipo bruto de Stack:

```
Stack< Integer > integerStack = new Stack( 10 );
```

Embora estas atribuições sejam permitidas o compilador gera avisos que importunam o programador para incentivá-lo a revisar seu código, mas não impedem que o código compile e funcione.

CORINGAS DE TIPO

Na Listagem 5 temos um método que soma os elementos de um *array* independente de seus tipos, este método precisa receber um *array* do tipo Number que é a superclasse de Integer e Double.

Listagem 5. Programa que soma os elementos de um ArrayList.

```
public class TotalNumbers {
   public static void main( String args[] ) {
      // cria, inicializa e gera saída de ArrayList de
      // números contendo Integer e Double e então
      // exibe o total dos elementos
      Number [] numbers = { 1, 2.4, 3, 4.1}; // Integer e Double

      ArrayList<Number> numberList = new ArrayList<Number>();
      for ( Number element : numbers)
         numberList.add( element ); // insere números na lista
         System.out.printf(“ Lista numberList contém: %s\n”,
            numberList );
         System.out.printf(“ Soma dos elementos: %.1f\n”,
            sum( numberList) );
   } // fim do main
```

```
   // calcula o total de elementos em ArrayList
   public statict double sum( ArrayList< Number > list ) {
      double total = 0; // inicializa o total
      // calcula a soma
      for ( Number element :list )
         total += element.doublevalue();
      return total;
   } // fim do método sum
} //fim da classe TotalNumbers
```

Dado ao fato que o método sum() pode somar elementos de um ArrayList de Number, talvez se espere que este método também funcione para ArrayList de Integer. Mas o método não funciona e o compilador emite a mensagem de erro:

```
sum(java.util.ArrayList<java.lang.Number>) in TotalNumbers
cannot be applied to (java.util.ArrayList<java.lang.Integer>)
```

Embora Number seja a superclasse de Integer, o compilador não considera o tipo parametrizado ArrayList< Number > como supertipo de Arraylist< Integer >.

O tipo coringa permite a criação de uma versão mais flexível do método sum() que possa somar os elementos de qualquer ArrayList que contenha elementos de qualquer subclasse de Number.

Os coringas permitem especificar parâmetros de métodos, valores de retorno e variáveis que atuam como supertipos de tipos parametrizados. O argumento tipo curinga é simbolizado por um ? (ponto de interrogação), que significa um tipo desconhecido.

Na Listagem 6 o curinga herda da classe Number, o que significa que o curinga tem um limite superior de Number.

Listagem 6. Programa para testar o coringa.

```
public class WildcardTest {
   public static void main( String args[] ) {
      // cria, inicializa e gera saída de ArrayList de Integer,
      // então exibe o total dos elementos
      Integer[] integers = { 1, 2, 3, 4, 5 };
      ArrayList<Integer> integerList = new ArrayList<Integer>();
      for (Integer element : integers)
         integerList.add( element ); // insere números na lista
      System.out.printf( “Lista integerList contém: %s\n”,
         integerList );
      System.out.printf( “Soma dos elementos: %.0f\n\n”,
         sum( integerList ) );

      // cria, inicializa e gera saída de ArrayList de Double,
```

```
      // então exibe o total dos elementos
      Double[] doubles = { 1.1, 3.3, 5.5 };
      ArrayList<Double> doubleList = new ArrayList<Double>();
      for (Double element : doubles)
         doubleList.add( element ); // insere números na lista
      System.out.printf( "Lista doubleList contém: %s\n",
         doubleList );
      System.out.printf( "Soma dos elementos: %.1f\n\n",
         sum( doubleList ) );

      // cria, inicializa e gera saída de ArrayList de Number
      // contendo Integer e Double e então exibe seu total
      Number [] numbers = { 1, 2.4, 3, 4.1}; // Integer e Double
      ArrayList<Number> numberList = new ArrayList<Number>();
      for ( Number element : numbers)
         numberList.add( element ); // insere números na lista
      System.out.printf( "Lista numberList contém: %s\n",
         numberList );
      System.out.printf(" Soma dos elementos: %.1f\n",
         sum( numberList) );
   } // fim do main

   // calcula o total de elementos em ArrayList
   public static double sum(ArrayList< ? extends Number > list){
      double total = 0; // inicializa o total
      // calcula a soma
      for ( Number element :list )
         total += element.doublevalue();
      return total;
   } // fim do método sum
} //fim da classe WildcardTest
```

Desta forma, o método sum() pode receber um argumento ArrayList que contém qualquer tipo de Number, como ArrayList<Integer>, ArrayList<Double> ou ArrayList<Number>, mostrando a versatilidade do uso do coringa.

CONSIDERAÇÕES FINAIS

Através dos exemplos de tipos, métodos e classes genéricas pudemos observar que os tipos genéricos vieram para aperfeiçoar o sistema de tipos estático do Java, eliminando quase todas as necessidades de *typecast*, deixando o código mais robusto, mais legível e mais fácil de ser testado.

Se existe uma modelagem formal do seu programa, o ideal e coletar aos seus diagramas para estudar onde os tipos genéricos poderiam contribuir para tornar o modelo mais robusto. Em alguns casos é possível que está informação já faça parte da modelagem.

REFERÊNCIAS BIBLIOGRÁFICAS

JANDL JR., P. **Java: Guia do Programador**. São Paulo: Novatec, 2007.

JCA COMMUNITY PROCESS. **JSR-14 (Java Generics)**. Disponível em: http://jcp.org/jsr/detail/14.jsp. Acessado em 30/09/2008.

PROGRAMMING METHODOLOGY GROUP. **PolyJ**. Disponível em: http://pmg.lcs.mit.edu/polyj/. Acessado em 30/09/2008.

SUN MICROSYSTEMS. **Generics Early Access**. Disponível em: http://developer.java.sun.com/developer/earlyAccess/adding_generics/. Acessado em 30/09/2008A.

_____. *Java Development Kit Documentation*. Disponível em: http://java.sun.com/j2se/1.5.0/download.jsp. Acessado em 30/09/2008B.