Módulo 2

Introdução à Programação II

Lição 13

Introdução à *Generics*

Versão 1.0 - Mar/2007

Autor
Rebecca Ong

Equipe
Joyce Avestro
Florence Balagtas
Rommel Feria
Rebecca Ong
John Paul Petines
Sun Microsystems
Sun Philippines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações:
http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Alexandre Mori
Alexis da Rocha Silva
Aline Sabbatini da Silva Alves
Allan Wojcik da Silva
André Luiz Moreira
Andro Márcio Correa Louredo
Antoniele de Assis Lima
Antonio Jose R. Alves Ramos
Aurélio Soares Neto
Bruno da Silva Bonfim
Bruno dos Santos Miranda
Bruno Ferreira Rodrigues
Carlos Alberto Vitorino de Almeida
Carlos Alexandre de Sene
Carlos André Noronha de Sousa
Carlos Eduardo Veras Neves
Cleber Ferreira de Sousa
Cleyton Artur Soares Urani
Cristiano Borges Ferreira
Cristiano de Siqueira Pires
Derlon Vandri Aliendres
Fabiano Eduardo de Oliveira
Fábio Bombonato
Fernando Antonio Mota Trinta
Flávio Alves Gomes
Francisco das Chagas
Francisco Marcio da Silva
Gilson Moreno Costa
Givailson de Souza Neves
Gustavo Henrique Castellano
Hebert Julio Gonçalves de Paula
Heraldo Conceição Domingues
Hugo Leonardo Malheiros Ferreira
Ivan Nascimento Fonseca
Jacqueline Susann Barbosa
Jader de Carvalho Belarmino
João Aurélio Telles da Rocha
João Paulo Cirino Silva de Novais
João Vianney Barrozo Costa
José Augusto Martins Nieviadonski
José Leonardo Borges de Melo
José Ricardo Carneiro
Kleberth Bezerra G. dos Santos
Lafaiete de Sá Guimarães
Leandro Silva de Morais
Leonardo Leopoldo do Nascimento
Leonardo Pereira dos Santos
Leonardo Rangel de Melo Filardi
Lucas Mauricio Castro e Martins
Luciana Rocha de Oliveira
Luís Carlos André
Luís Octávio Jorge V. Lima
Luiz Fernandes de Oliveira Junior
Luiz Victor de Andrade Lima
Manoel Cotts de Queiroz
Marcello Sandi Pinheiro
Marcelo Ortolan Pazzetto
Marco Aurélio Martins Bessa
Marcos Vinicius de Toledo
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Mauricio Azevedo Gamarra
Mauricio da Silva Marinho
Mauro Cardoso Mortoni
Mauro Regis de Sousa Lima
Namor de Sá e Silva
Néres Chaves Rebouças
Nolyanne Peixoto Brasil Vieira
Paulo Afonso Corrêa
Paulo José Lemos Costa
Paulo Oliveira Sampaio Reis
Pedro Antonio Pereira Miranda
Pedro Henrique Pereira de Andrade
Renato Alves Félix
Renato Barbosa da Silva
Reyderson Magela dos Reis
Ricardo Ferreira Rodrigues
Ricardo Ulrich Bomfim
Robson de Oliveira Cunha
Rodrigo Pereira Machado
Rodrigo Rosa Miranda Corrêa
Rodrigo Vaez
Ronie Dotzlaw
Rosely Moreira de Jesus
Seire Pareja
Sergio Pomerancblum
Silvio Sznifer
Suzana da Costa Oliveira
Tásio Vasconcelos da Silveira
Thiago Magela Rodrigues Dias
Tiago Gimenez Ribeiro
Vanderlei Carvalho Rodrigues Pinto
Vanessa dos Santos Almeida
Vastí Mendes da Silva Rocha
Wagner Eliezer Roncoletta

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Regina Mariani -** Coordenadora responsável pela parte jurídica
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

A versão 5 do Java trouxe grandes avanços na programação. Inclui extensões significantes à sintaxe da linguagem. A mais visível delas foi a adição de *Generics*. Nessa lição iremos conhecer os conceitos básicos relacionados a Generics.

Ao final desta lição, o estudante será capaz de:

- Enumerar os benefícios de *Generics*
- Declarar classes utilizando *Generics*
- Utilizar *Generics* limitados
- Declarar métodos utilizando *Generics*
- Usar coleções Java com *Generics*

2. Por que *Generics*?

Um dos maiores problemas na linguagem de programação Java é a necessidade constante de fazer *typecast* ou *casting* (conversões de tipo explícitas) em expressões para tipos de dados mais específicos do que seus tipos estáticos. Por exemplo, um objeto *ArrayList* permite adicionarmos objetos de qualquer tipo de referência à coleção, mas quando recuperarmos estes elementos, precisaremos fazer um *typecast* nos objetos para o tipo de referência específico à nossa necessidade. O *casting* é um perigo em potencial para gerar uma *ClassCastException*. Ele também torna nossos códigos mais poluídos e, portanto, menos legíveis. Além disso, o *casting* efetivamente destrói os benefícios de uma linguagem com os tipos fortemente definidos, uma vez que anula a segurança trazida pela checagem embutida de tipos.

O principal objetivo da adição de *Generics* ao Java é solucionar este problema. Tipos *Generics* permite que uma única classe trabalhe com uma grande variedade de tipos. Eles são uma forma natural de se eliminar a necessidade do *cast*.

Primeiro iremos considerar um objeto *ArrayList* e ver como os tipos *Generics* ajudariam a melhorar nosso código. Um objeto *ArrayList* possui a capacidade de armazenar elementos de qualquer tipo de referência em uma coleção de objetos. Uma instância de *ArrayList*, entretanto, sempre força a realizar um *casting* nos objetos que recuperamos da coleção. Considere a seguinte linha de instrução:

```
String myString = (String) myArrayList.get(0);
```

A versão utilizando *Generics*, da classe *ArrayList* foi desenvolvida para trabalhar nativamente com qualquer tipo de classe. Ao mesmo tempo, ela também preserva os benefícios da checagem de tipos. Podemos eliminar a necessidade do *casting* no elemento que obtemos da coleção e ter a seguinte linha de instrução:

```
String myString = myArrayList.get(0);
```

Embora o *casting* já tenha sido removido, isso não significa que possamos atribuir qualquer tipo ao valor de retorno do método *get* e eliminar o *casting* também. Ao atribuir outra coisa que não uma *String* à saída do método *get*, obteremos um erro de equiparação de tipos em tempo de compilação tal como essa mensagem:

```
found: java.lang.String
required: java.lang.Integer
Integer data = myArrayList.get(0);
```

Para se ter uma idéia de como *Generics* é usado, antes de entrarmos em mais detalhes, considere o seguinte fragmento de código:

```
ArrayList <String> genArrList = new ArrayList <String>();
genArrList.add("A generic string");
String myString = genArrList.get(0);
JoptionPane.showMessageDialog(this, myString);
```

Analisando estas instruções, observamos a palavra <String> aparecendo imediatamente após a referência do tipo *ArrayList*. Podemos interpretar esta como o primeiro comando da criação de uma instância de uma versão da classe *ArrayList* utilizando *Generics,* e esta versão contém objetos do tipo String. *genArrList* está **preso** ao tipo *String*. Portanto, "prender" um *Integer* ou outro tipo não *String* ao resultado da função *get* seria ilegal. A próxima instrução não seria válida:

```
int myInt = genArrList.get();
```

3. Declarando uma Classe Utilizando *Generics*

Para que o fragmento de código anterior funcione, deveríamos ter definido uma versão da classe *ArrayList* que utilize *Generics*. Felizmente, a versão mais recente do Java fornece aos usuários versões que fazem uso de *Generics* para todas as implementações de *Collection*. Nessa seção, aprenderemos como declarar sua própria classe utilizando *Generics*.

Ao invés de uma longa discussão sobre como declarar uma classe utilizando *Generics*, veremos um exemplo:

```
class BasicGeneric <A> {
    private A data;
    public BasicGeneric(A data) {
        this.data = data;
    }
    public A getData() {
        return data;
    }
}
public class GenericDemo {
    public String method(String input) {
        String data1 = input;
        BasicGeneric <String> basicGeneric = new
                BasicGeneric <String>(data1);
        String data2 = basicGeneric.getData();
        return data2;
    }
    public Integer method(int input) {
        Integer data1 = new Integer(input);
        BasicGeneric <Integer> basicGeneric = new
                BasicGeneric <Integer>(data1);
        Integer data2 = basicGeneric.getData();
        return data2;
    }
    public static void main(String args[]) {
        GenericDemo sample = new GenericDemo();
        System.out.println(sample.method("Some generic data"));
        System.out.println(sample.method(1234));
    }
}
```

Na execução desta classe será mostrada a seguinte saída:

```
Some generic data
1234
```

Iremos agora passar pelas instruções que possuem a sintaxe que utilizam *Generics*.

Para a declaração da classe *BasicGeneric*:

```
class BasicGeneric <A>
```

o nome da classe é seguido por um par de sinais "menor que" e "maior que" envolvendo a letra maiúscula A: <A>. Isto é chamado de um parâmetro de tipo. Os usos desses sinais "menor que" e "maior que" indicam que a classe declarada é uma classe que utiliza Generics. Isso significa que a classe não trabalha com nenhuma referência a um tipo específico. Por isso, observe que o atributo da classe foi declarado para ser do tipo A:

```
private A data;
```

Essa declaração especifica que o atributo *data* é de um tipo *Generic*, e depende do tipo de dado com que o objeto *BasicGeneric* for desenvolvido para trabalhar.

Quando declarar uma instância da classe, deve ser especificada a referência ao tipo com que se deseja trabalhar. Por exemplo:

```
BasicGeneric <String> basicGeneric = new BasicGeneric <String>(data1);
```

A sintaxe <*String*> após a declaração de *BasicGeneric* especifica que essa instância da classe irá trabalhar com variáveis do tipo *String*.

Podemos trabalhar com variáveis do tipo *Integer* ou qualquer outro tipo de referência. Para trabalhar com um *Integer*, o fragmento do código teria a seguinte instrução:

```
BasicGeneric <Integer> basicGeneric = new BasicGeneric <Integer>(data1);
```

Consideremos a declaração para o método *getData*:

```
public A getData() {
   return data;
}
```

O método *getData* retorna um valor do tipo *A*, que é um tipo *Generic*. Isso significa que o método terá um tipo de dado em tempo de execução, ou mesmo em tempo de compilação. Depois de declarar um objeto do tipo *BasicGeneric*, *A* será "preso" a um tipo de dado específico. Essa instância atuará como se tivesse sido declarada para ter esse tipo de dado específico, e apenas esse, desde o início.

No seguinte código, duas instâncias da classe *BasicGeneric* foram criadas.

```
BasicGeneric <String> basicGeneric = new BasicGeneric <String>(data1);
String data2 = basicGeneric.getData();

BasicGeneric <Integer> basicGeneric = new BasicGeneric <Integer>(data1);
Integer data2 = basicGeneric.getData();
```

Note que a criação de uma instância de uma classe que utiliza *Generics* é bem similar à criação de uma classe normal, exceto pelo tipo de dado específico dentro dos sinais <> logo após o nome do construtor. Essa informação adicional indica o tipo de dado com que você trabalhará para essa instância da classe *BasicGeneric* em particular. Depois de criada a instância, é possível agora acessar os elementos da classe através dela. Não existe mais a necessidade do *typecast* no valor do retorno do método *getData*, uma vez que já foi decidido que ele irá trabalhar com um tipo de dado de referência específico.

3.1. Limitação "Primitiva"

Uma limitação de *Generics* em Java, é que eles são restritos a tipos de referência (Objetos) e não funcionarão com tipos primitivos.

Por exemplo, o comando a seguir seria ilegal, uma vez que *int* é um tipo de dado primitivo.

```
BasicGeneric <int> basicGeneric = new
                                    BasicGeneric <int>(data1);
```

Você terá que encapsular os tipos primitivos primeiro, antes de usá-los como argumentos *Generics*.

3.2. Compilando Generics

Para compilar códigos fonte Java com *Generics* usando JDK (v. 1.5.0), utilize a seguinte sintaxe:

```
javac -version -source "1.5" -sourcepath src -d classes
src/SwapClass.java
```

onde *src* refere-se à localização do código fonte java, enquanto *class* refere-se à localização onde o arquivo da classe será gravado.

Aqui está um exemplo:

```
javac -version -source "1.5" -sourcepath c:\temp -d c:\temp
c:/temp/SwapClass.java
```

4. Generics Limitados

No exemplo mostrado anteriormente, os parâmetros de tipo da classe *BasicGeneric* podem ser de qualquer tipo de dado de referência. Há casos, entretanto, onde você quer restringir os tipos em potencial usados em instanciações de uma classe do tipo *Generics*. Java permite limitar o conjunto de possíveis tipos utilizados como argumento na instanciação de classes desse tipo, para um conjunto de subtipos de um determinado tipo "amarrado" à classe.

Por exemplo, podemos definir uma classe *ScrollPane* que utiliza *Generics* que serviria como um molde para um *Container* comum com a funcionalidade da barra de rolagem. Em tempo de execução, o tipo da instância dessa classe será, geralmente, uma subclasse de *Container*, mas o tipo estático ou geral é simplesmente *Container*.

Para limitar as instanciações de tipo de uma classe, nós usamos a palavra-chave *extends* seguida pela classe que limita (ou restringe) o tipo *Generics* como parte de um parâmetro de tipo.

O exemplo a seguir limita as instanciações de tipo da classe *ScrollPane* para subtipos da classe *Container*.

```
class ScrollPane <MyPane extends Container> {
   ...
}
class TestScrollPane {
   public static void main(String args[]) {
      ScrollPane <Panel> scrollPane1 = new ScrollPane <Panel>();
      // O comando seguinte é ilegal
      ScrollPane <Button> scrollPane2 = new ScrollPane <Button>();
   }
}
```

A instanciação de *scrollPane1* é válida, uma vez que *Panel* é uma subclasse de *Container*, enquanto a criação de *scrollPane2* causaria um erro em tempo de compilação, uma vez que *Button* não é uma subclasse de *Container*.

Usar *Generics* limitados nos dá um adicional, que é a checagem estática de tipos. Como resultado, nós temos a garantia que toda instanciação de um tipo *Generics* respeita os limites (ou restrições) que atribuímos a ele.

Uma vez que asseguramos que toda instância de tipo é uma subclasse do limite atribuído, podemos chamar, de forma segura, qualquer método encontrado no tipo estático do objeto. Se não tivéssemos colocado nenhum limite explícito no parâmetro, o limite default seria *Object*. Isso significa que não podemos invocar métodos em uma instância do limite que não apareçam na classe *Object*.

5. Declarando Métodos *Generics*

Além de declarar classe *Generics*, também podemos declarar métodos *Generics*. Estes métodos são chamados polimórficos, e são definidos para serem métodos parametrizados pelo tipo.

Parametrizar métodos é útil quando queremos realizar tarefas onde as dependências de tipo entre os argumentos e o valor de retorno são *Generics*, mas não depende de nenhuma informação do tipo da classe, e mudará a cada chamada ao método.

Por exemplo, suponha que queremos adicionar um método *make* em uma classe *ArrayList*. Este método estático admitiria um único argumento, que seria o único elemento do objeto *ArrayList*. Para fazer com que nosso *ArrayList* receba qualquer tipo de elemento, o argumento e o retorno do método *make* deve utilizar Generics.

Para declarar tipos *Generics* no nível de métodos, considere o exemplo a seguir:

```
class Utilities {
   /* T extends Object implicitamente */
   public static <T> ArrayList<T> make(T first) {
      return new ArrayList<T>(first);
   }
}
```

Java também utiliza um mecanismo de inferência de tipos, para inferir automaticamente os tipos dos métodos polimórficos baseado no tipo dos argumentos. Isso diminui o excesso de palavras, e a complexidade nas invocações de métodos.

Para construir uma nova instância de *ArrayList<Integer>*, nós simplesmente teríamos o seguinte comando:

```
Utilities.make(Integer(0));
```

6. *Generics* e Coleções Java

Uma discussão sobre métodos da interface *Collection* já foi mostrada em lições anteriores. Entretanto, no JDK 5.0, esses métodos foram atualizados para acomodar *Generics*. A tabela a seguir mostra os métodos do JDK 5.0 equivalentes àqueles que você estudou anteriormente.

Métodos da interface Collection
`public boolean add(E o)`
Insere o elemento específico *o* nessa coleção. Retorna *true* se *o* foi adicionado com sucesso à coleção.
`public void clear()`
Remove todos os elementos dessa coleção.
`public boolean remove(Object o)`
Remove uma única instância de *Object o* dessa coleção, se ele estiver presente. Retorna *true* se *o* foi encontrado e removido da coleção.
`public boolean contains(Object o)`
Retorna true se essa coleção contém o *Object o*.
`public boolean isEmpty()`
Retorna true se essa coleção não contém nenhum objeto ou elemento.
`public int size()`
Retorna o número de elementos nessa coleção.
`public Iterator<E> iterator()`
Retorna um iterator que nos permite acessar elementos da coleção.
`public boolean equals(Object o)`
Retorna true se o *Object o* é igual a essa coleção.
`public int hashCode()`
Retorna o valor do *hash code* (i.e., o ID) para essa coleção. Mesmos objetos ou coleções têm o mesmo valor de *hash code* ou ID.

Table 1: métodos da interface Collection

Aqui está um exemplo de uma classe que não utiliza a versão *Generics* para a criação de uma lista, observe que podemos incluir sem problemas um objeto String entre os outros elementos

```
import java.util.*;

class GenericDemo {
    public static void main(String args[]) {
        List list = new ArrayList();
        list.add(new Integer(1));
        list.add(new Integer(2));
        list.add("String");
        System.out.println(list);
        Collections.sort(list);
        System.out.println(list);
    }
}
```

Executar esta classe é gerada a seguinte saída:

```
[1, 2, String]
Exception in thread "main" java.lang.ClassCastException:
java.lang.String
        at java.lang.Integer.compareTo(Integer.java:35)
        at java.util.Arrays.mergeSort(Arrays.java:1156)
        at java.util.Arrays.sort(Arrays.java:1080)
        at java.util.Collections.sort(Collections.java:117)
        at GenericDemo.main(GenericDemo.java:10)
Java Result: 1
```

Pelo simples motivo que o Java não sabe como ordenar uma String juntamente com dois

objetos inteiros, agora vejamos a mesma classe utilizando agora o Generics:

```
import java.util.*;

class GenericDemo {
    public static void main(String args[]) {
        List<Integer> list = new ArrayList<Integer>();
        list.add(new Integer(1));
        list.add(new Integer(2));
        list.add("String");
        System.out.println(list);
        Collections.sort(list);
        System.out.println(list);
    }
}
```

O que acontecerá será um erro de compilação na linha 8, ou seja, o Generics bloqueia os elementos estranhos a lista não permitindo que esta tenha elementos diversos fora o especificado.

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.