Capítulo 12

Java Generics

Co-autor: Bruno Pandolfi

Já vimos que o J2SE 5.0 introduz várias mudanças significativas na linguagem. A principal modificação realizada é a implementação de tipos parametrizados, conhecidos como generics. Algumas linguagens também oferecem as vantagens trazidas por generics. Os programadores de C++ já viram algo semelhante ao trabalhar com templates. Ao longo deste capítulo, será observado que há semelhanças e diferenças importantes entre os mecanismos de cada linguagem.

12.1 – Usando Generics

Generics oferece ao programador uma abstração sobre tipos de dados. Essa abstração é bastante útil quando se trata do uso de objetos do tipo ArrayList, LinkedList, e seus semelhantes da hierarquia Collection. O exemplo seguinte ilustra como se procedia antes de generics:

```
List myIntList = new LinkedList(); //1
myIntList.add(new Integer(0)); //2
Integer x = (Integer) myIntList.iterator().next(); //3
```

Exemplo 12. 1 – Sem Tipos Parametrizados

O cast na linha 3 parece desnecessário. Afinal, o programador sabe, a priori, qual tipo de dados sua lista irá conter. Entretanto, o cast é essencial, pois o compilador apenas pode garantir que um Object será retornado pelo método next do iterador. Para assegurar que a atribuição é válida, o compilador requer que o cast seja feito.

A principal idéia por trás de generics é restringir o conteúdo de uma lista a apenas objetos de um tipo escolhido. Observe, a seguir, o trecho de código agora escrito usando generics:

```
List<Integer> myIntList =
    new LinkedList<Integer>(); // 1'
myIntList.add(new Integer(0)); //2'
Integer x = myIntList.iterator().next(); // 3'
```

Exemplo 12. 2 – Com Tipos Parametrizados

Preste atenção na declaração de myIntList. Ela especifica que não se trata de uma lista qualquer, mas sim de uma lista para inteiros, como se nota em List<Integer>. Desta forma é dito que List é uma interface genérica que requer um tipo como parâmetro – neste caso, o tipo passado foi Integer. Também é especificado o tipo no momento da construção do objeto. Note também que o cast sumiu da linha 3'.

É possível imaginar que o cast foi apenas movido da linha 3' para a linha 1'. Entretanto, há uma grande diferença. Agora, o compilador pode checar os tipos e validar as operações em tempo de compilação, e o programador não precisa mais se preocupar com casts em cada atribuição. Desta maneira, tem-se um significativo aumento da confiabilidade e segurança do código. O efeito final, especialmente no desenvolvimento de grandes aplicações, é o aumento da legibilidade, redigibilidade e da robustez.

12.2 – Definindo Generics

Observe a seguir um trecho do código de definição das interfaces List e Iterator do pacote java.util.

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}
public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

Exemplo 12. 3 – Definição de Generics

Tudo deveria ser familiar, exceto pelos <E> espalhados pelo código. Trata-se da definição de parâmetros formais de tipo. Na seção anterior foi visto como se faz a declaração de variáveis usando tipos genéricos (tais como em List<Integer>). Quando essas declarações são invocadas, todas as ocorrências dos parâmetros formais E são substituídas pelo parâmetro real Integer.

É plausível imaginar que a declaração List<Integer> requer uma prévia implementação semelhante a:

```
public interface IntegerList {
    void add(Integer x)
    Iterator<Integer> iterator();
}
```

Exemplo 12. 4 – Definição de Generics

Essa noção intuitiva é útil, porém errônea. É útil porque o comportamento de vários métodos de tipos genéricos se assemelha em muito a esta idéia. É errônea porque a declaração nunca é expandida desta forma. Não há códigos diferentes gerados para atender especificamente a cada instanciação de List. A definição de um tipo genérico é compilada uma única vez e colocada em um arquivo .class comum, da mesma maneira que as demais classes e interfaces são compiladas. Esse único código é usado para todas as instanciações desse tipo parametrizado. Os programadores de C++ sabem que essa é uma diferença significativa entre generics e templates.

Tipos parametrizados são tratados analogamente aos demais parâmetros por métodos ou construtores. Do mesmo modo que um método possui parâmetros formais de valor, que indicam o tipo de dados com que o método irá trabalhar, uma definição de tipo parametrizado possui parâmetros formais de tipo. Quando um método é invocado, os parâmetros formais são substituídos pelos parâmetros reais, e o método é executado. Quando uma declaração de um tipo genérico é invocada, o parâmetro formal de tipo é substituído pelo parâmetro real, escolhido pelo usuário.

Há algumas convenções quando se definem tipos parametrizados. É evitado o uso de nomes significativos para os parâmetros formais de tipo. O mais comum é atribuir uma letra maiúscula para cada parâmetro (a maioria das listas e demais containers utiliza a letra E, em referência a Elemento).

12.3 – Generics e Subtipos

Verifique o código a seguir. Trata-se de uma implementação válida?

```
List<String> ls = new ArrayList<String>(); //1
List<Object> lo = ls; //2
```

A linha 1 é correta, devido ao que acabou de ser discutido. O problema está na linha 2. Ao atribuir a lista de String a uma lista de Object, o programador desavisado poderia considerar que não há problema, uma vez que a classe Object é superclasse das demais classes de Java. Porém, observe a continuação do programa:

```
lo.add(new Object()); // 3
String s = ls.get(0); // 4: atribui Object a String
```

Fazendo a amarração da lista lo à lista ls, ls passará a não conter somente String, pois lo permite a adição de qualquer objeto. Esse erro é apontado pelo compilador. O motivo para o equívoco é que a idéia de Herança de características não se aplica aos tipos parametrizados. Em geral, se Classe1 é uma subclasse de ClasseMãe, List<Classe1> não é uma subclasse de List<ClasseMãe>.

12.3.1 – Wildcards (Coringas)

Considere o problema de escrever uma rotina para imprimir os objetos de uma coleção. Observe a seguir como era feito antes da versão 1.5 de Java.

```
void printCollection(Collection c) {
    Iterator i = c.iterator();
    for (k = 0; k < c.size(); k++) {
        System.out.println(i.next());
    }
}
```

Veja abaixo a nova sintaxe do loop for e uma implementação intuitiva usando Java generics.

```
void printCollection(Collection<Object> c) {
    for (Object e: c) {
        System.out.println(e);
    }
}
```

Esta tentativa é muito menos útil que a anterior, pois ela pode ser invocada apenas com coleções de Object como parâmetro (as quais, como já discutido, não constituem a superclasse de todas as coleções). Para se referir a coleções de qualquer tipo é preciso escrever Coleção<?>. Esta estrutura é chamada de wildcards (coringas), por razões óbvias. Sendo assim, a implementação correta para a rotina desejada é:

```
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

Exemplo 12. 5 – Uso de Wildcards

Note que essa rotina ainda lê os elementos de c e os atribui a uma variável Object. Isso é válido, uma vez que independentemente do tipo da coleção, ele sempre será um subtipo de Object. Entretanto, é inválido inserir qualquer objeto em uma coleção definida para um tipo desconhecido, tal como se faz no exemplo a seguir:

```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // erro em tempo de compilação
```

Uma vez que não se sabe o tipo de objetos que c armazena, não se pode garantir a validade de nenhuma inserção. A exceção é a inserção de null, o qual é membro de todas as classes.

É bom frisar a interessante vantagem conseguida ao se usar wildcards. Eles permitem generalizar a aplicabilidade de métodos sem que se saiba, de antemão, qual o tipo de dados que a coleção armazena. É possível recuperar e trabalhar com os elementos de qualquer lista atribuindo-os a uma variável Object (a justificativa para tal foi dada há pouco). Porém, não se pode fazer nenhuma inserção em coleções de tipo desconhecido.

12.3.2 – Wildcards Limitados

Considere uma simples aplicação gráfica que consegue desenhar formas simples, tais como retângulos e círculos. Para representar essas formas neste programa, usou-se a seguinte hierarquia de classes:

```
public abstract class Forma {
    public abstract void desenhar();
}
public class Circulo extends Forma {
    private int x, y, raio;
    public void desenhar() {
       System.out.println("Circulo");
    }
}
public class Retangulo extends Forma {
    private int x, y, largura, altura;
    public void desenhar() {
       System.out.println("Forma");
    }
}
```

Qualquer desenho deverá conter uma ou mais formas. Assume-se que elas estarão contidas numa lista. Desta forma, é prudente que se disponibilize um método que imprima todas as formas de um desenho:

```
public void desenhaTudo(List<Forma> formas) {
    for (Forma f: formas) {
        f.desenhar();
    }
```

```
}
```

Entretanto, as regras de tipos genéricos afirmam que apenas listas de Forma podem ser desenhadas. Não se poderá usar essa rotina, por exemplo, para desenhar um List<Retangulo>. Isso é um tanto quanto indesejável, pois tudo o que se faz é ler formas de uma lista e desenhá-las. Neste caso, não deveria haver empecilho para aplicar a rotina a List<Retangulo>.

O que se busca, na verdade, é um método que possa ser empregado para a classe Forma e todas as suas subclasses. Isso é conseguido usando a estrutura seguinte:

```
public void desenhaTudo(List<? extends Forma> formas){
   ...
}
```

Exemplo 12. 6 – Uso de Wildcards Limitados

Há uma pequena, porém importante, diferença. Agora, pode-se usar desenharTudo para qualquer classe que estenda Forma, ou ela mesma, inclusive. List<? extends Forma> é um exemplo de wildcard limitado. Forma é dito ser o limite superior do wildcard.

Existe, porém, como em quase tudo que gera vantagens, um preço a pagar pela flexibilidade proporcionada pelos wildcards. Veja no código a seguir um possível erro ao se usar wildcards na programação:

```
public void addRetangulo(List<? extends Forma> formas)
{
    formas.add(0, new Retangulo()); // erro em tempo
de compilação
}
```

O parâmetro formal para a rotina anterior é uma lista de "? extends Forma", ou seja, de um subtipo desconhecido de Forma. Neste cenário, é impossível garantir a validade da inserção de um Retangulo, uma vez que o parâmetro real poderia ser List<Circulo>. Esse revés também ocorre com os wildcards não limitados, conforme comentado no exemplo anterior sobre imprimir os objetos de uma coleção.

12.4 – Métodos Genéricos

Imagine agora que seja necessário escrever um método que converta um dado array em uma coleção. Uma primeira tentativa poderia ser:

```
static void fromArrayToCollection(Object[] a,
                                  Collection<?> c) {
    for (Object o : a) {
        c.add(o); // erro de compilação
    }
}
```

No exemplo seria um engano usar Collection<Object> como atributo da rotina. O uso de wildcards também não dará resultado, pois é preciso lembrar que não se pode fazer inserções em coleções de tipo desconhecido.

A maneira correta de abordar esse problema é pelo uso de métodos genéricos. Assim como tipos parametrizados, métodos podem ser generalizados para um ou mais parâmetros de tipo. Veja no exemplo a seguir.

```
static <T> void fromArrayToCollection(T[] a,
                    Collection<T> c) {
   for (T o : a) {
      c.add(o); // correto!
   }
}
```

Exemplo 12. 7 – Métodos Genéricos

Pode-se chamar essa rotina passando qualquer coleção que armazene valores do tipo ou de um subtipo dos elementos do array.

```
Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();
fromArrayToCollection(oa, co);// T como Object
String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();
fromArrayToCollection(sa, cs);// T como String
fromArrayToCollection(sa, co);// T como Object
Integer[] ia = new Integer[100];
Float[] fa = new Float[100];
Number[] na = new Number[100];
Collection<Number> cn = new ArrayList<Number>();
fromArrayToCollection(ia, cn);// T como Number
fromArrayToCollection(fa, cn);// T como Number
fromArrayToCollection(na, cn);// T como Number
fromArrayToCollection(na, co);// T como Object
fromArrayToCollection(na, cs);// erro de compilação
```

Exemplo 12. 8 – Usando Métodos Genéricos

Note que não é preciso passar o parâmetro real de tipo para o método. O compilador infere o tipo automaticamente, baseado no tipo dos parâmetros reais de valor. Geralmente, será inferido o tipo mais específico que garanta a validade da chamada.

É importante saber as situações nas quais se devem usar wildcards ou métodos genéricos, e as vantagens e desvantagens de cada abordagem. Wildcards foram criados para dar maior flexibilidade ao se generalizar tipos e subtipos. Quando a idéia principal for empregar polimorfismo, então os wildcards devem ser usados. Métodos genéricos, por sua vez, foram desenvolvidos para exprimir dependências entre os parâmetros formais de tipo presentes em uma definição genérica (de classe ou de método). Observe o exemplo seguinte, que trata de um método genérico que usa wildcards:

```
class Collections {
    public static <T> void copy(List<T> dest,
       List<? extends T> src){...}
}
```

Exemplo 12. 9 – Métodos Genéricos com Wildcards

A assinatura de copy expressa a dependência entre os tipos das listas que são recebidas como parâmetros de entrada. Neste caso, especificamente, o wildcard serviu para informar que o elemento da lista fonte da cópia deve ser de algum subtipo do elemento da lista destino (ou do próprio tipo do elemento da lista destino).

Uma possibilidade seria escrever a rotina usando somente a abordagem por métodos genéricos:

```
class Collections {
  public static <T, S extends T> void copy(
     List<T> dest, List<S> src){...}
}
```

Esta abordagem, apesar de funcionar corretamente, deve ser evitada. O parâmetro de tipo S é desnecessário, pois é usado apenas na definição de um parâmetro formal parametrizado (a lista fonte src), e nada mais depende dele. Isso é um indicativo de que se deve usar wildcards. O uso de wildcards é mais conciso e mais claro do que a declaração explícita de parâmetros de tipo e deve, portanto, ter preferência durante a escrita de programas.

Novamente é importante considerar os padrões de notação empregados quando se utiliza generics. Usa-se T para indicar tipo, sempre que não houver nada mais específico para forçar a escolha de outro nome. Esse é o caso, normalmente, da implementação de métodos genéricos. Se houver mais de um parâmetro genérico, geralmente usam-se outras letras vizinhas a T no alfabeto. Se o método genérico aparece dentro de uma classe genérica, é interessante usar os padrões dessa classe para nomear os parâmetros genéricos.

12.5 – Generics e Códigos Legados

No mundo real, nem todas as pessoas estão ou estiveram usando a última versão de Java recentemente. O que se quer dizer é que há centenas de milhares de linhas de código escritas em diversas versões de Java e elas não serão atualizadas da noite para o dia. Por isso é necessário saber como é possível usar generics juntamente com esses códigos antigos (conhecidos com códigos legados).

12.5.1 – Usando Códigos Legados em Códigos Genéricos

Como um exemplo, assuma que se quer usar o pacote br.com.MorteIndolor.coisas, que constitui o inventário da fábrica de ferramentas Morte Indolor. Partes do código são mostradas a seguir:

```
package br.com.MorteIndolor.coisas;
public interface Parte { ...}
public class Inventario {
/**
* Inventario é um conjunto de instrumentos
* Um instrumento é caracterizado por um dado nome, e
* consiste de um conjunto de partes. Todos os
* elementos da coleção de partes devem implementar a
* interface "Parte".
**/
   public static void addInstrumento(String nome,
      Collection partes) {...}
```

```
   public static Instrumento getInstrumento(
      String nome) {...}
}
public interface Instrumento {
   Collection getPartes(); //Retorna coleção de partes
}
```

Exemplo 12. 10 – Código Legado

Agora se deseja adicionar um novo código que usa a API acima. Seria bom assegurar que addInstrumento sempre será chamado com os tipos corretos de argumento, isto é, que a coleção passada seja, de fato, uma coleção de Parte. Pois foi justamente para isso que os wildcards foram criados.

```
package br.com.minhaEmpresa.inventario;
import br.com.MorteIndolor.coisas.*;
public class  Lamina implements Parte { ...}
public class Suporte implements Parte { ...}
public class Main {
  public static void main(String[] args) {
     Collection<Parte> c = new ArrayList<Parte>();
     c.add(new Suporte()) ;
     c.add(new Lamina());
     Inventario.addInstrumento("Guilhotina", c);
     Collection<Parte> k =
         Inventario.getInstrumento(
              "Guilhotina").getPartes();
  }
}
```

Exemplo 12. 11 – Usando Código Legado no Código Genérico

Quando addInstrumento é chamado, ele espera que o segundo parâmetro seja do tipo Collection. O argumento real passado é Collection<Parte>. Isso funciona, mas por quê? A maioria das coleções não contém objetos do tipo Parte, e, em geral, o compilador não tem como saber a qual tipo de coleção um objeto Collection está se referindo.

Num código genérico adequado, Collection deveria sempre estar acompanhado de um parâmetro de tipo. Quando um Collection é usado sem seu parâmetro de tipo, ele é chamado de raw type (tipo natural).

A noção intuitiva da maioria das pessoas diria que Collection trata de uma Collection<Object>. Entretanto, como foi visto, não é seguro passar uma Collection<Parte> onde uma Collection<Object> for requerido. Seria mais sensato imaginar que Collection deve ser encarado como um wildcard de tipo desconhecido, ou seja, como Colletion<?>.

Mas também não está correto pensar desta forma! Considerando a chamada a getParts(), a qual retorna uma Collection. Esse retorno é associado a k, que é uma Collection<Parte>. Se o resultado fosse tratado como Collection<?>, a atribuição estaria errada.

Na realidade, a atribuição é legal. Todavia, ela gera na compilação um alerta de operação não verificada. O aviso é necessário, pois o fato é que o compilador não pode garantir a validade das atribuições. Não há como checar o código antigo para descobrir se a coleção

retornada por getInstrumento() é uma Collection<Parte>. O tipo usado no código importado é Collection, e pode receber, legalmente, coleções de qualquer tipo.

Teoricamente falando, esse impasse deveria ser considerado um erro. Mas na prática, se o código genérico vai chamar o código antigo, isso deve ser permitido. Deve ser bastante, para o programador, aceitar que a documentação de getPartes() garanta o retorno de uma coleção de partes, mesmo que a assinatura do método não assegure nada sobre isso.

Assim sendo, pode-se considerar os tipos naturais semelhantes aos wildcards. Porém, eles não garantem uma checagem de tipos rigorosa. Esta é uma opção deliberada dos projetistas de Java, para tornar possível mesclar códigos genéricos com códigos pré-existentes de versões anteriores.

Chamar códigos antigos de dentro de códigos genéricos será inerentemente inseguro. Ao misturar esses códigos, toda a garantia de checagem de tipos fornecida por generics é quebrada. No momento há muito mais códigos não-genéricos do que os que fazem uso de generics, mas certamente haverá cada vez mais situações em que ambos os códigos deverão trabalhar em conjunto. Quando essa mistura acontecer, o programador deve ficar atento aos avisos e pensar profundamente nos motivos que geraram cada alerta de "operação não-verificada".

A partir de agora, exibir-se-á o que acontece quando um erro gerado em uma atribuição não verificada não é corrigido pelo programador. Será visto, ainda, um pouco do funcionamento do compilador. Comece observando este exemplo:

```
public String brecha(Integer x) {
    List<String> ys = new LinkedList<String>();
    List xs = ys;
    xs.add(x); // alerta "não verificado"
    return ys.iterator().next();
}
```

Exemplo 12. 12 – Erro de Tipo em Código Genérico

Essa rotina possui uma atribuição de uma lista de String a uma lista simples definida como nas versões antigas da linguagem. É inserido um Integer na lista e uma String é retirada para ser retornada pelo método. Isso está claramente incorreto. Se o alerta for ignorado e o código for executado, ele falhará exatamente no ponto onde se tenta usar o tipo errado. Em tempo de execução, o código se comporta como:

```
public String brecha(Integer x) {
    List ys = new LinkedList;
    List xs = ys;
    xs.add(x);
    return (String) ys.iterator().next(); // erro
}
```

Exemplo 12. 13 – Erro de Tipo em Código Equivalente

Quando ele retira um elemento da lista e tenta tratá-lo como String, uma exceção ClassCastException será lançada, pois o elemento retirado é um Integer, em vez de String. A razão para isso é que generics é implementado por Java quase como uma tradução de código para código, na qual o código genérico é convertido para uma versão não-genérica.

O principal resultado dessa abordagem é que a integridade da máquina virtual Java nunca é colocada em risco, mesmo quando um alerta de "não-verificado" for ignorado. Esse

processo praticamente apaga toda a informação genérica de tipos. Por exemplo, List<String> é convertido em List. Todas as variáveis restantes definidas genericamente são convertidas para o tipo limite superior (normalmente Object). Além disso, casts são adicionados (como na última linda de brecha) para assegurar a validade das atribuições. Maiores detalhes sobre esses processos estão fora do escopo desse capítulo. Porém, o que foi apresentado é bastante próximo do que acontece na realidade.

12.5.2 – Usando Generics em Códigos Legados

Leve em conta agora o caso inverso. Imagine que a companhia Morte Indolor escolheu migrar seu código para uma versão que use generics, mas alguns de seus clientes ainda não o fizeram. Observe como o novo código está escrito:

```
package br.com.MorteIndolor.coisas;
public interface Parte { ...}
public class Inventario {
    public static void addInstrumento(String nome,
       Collection<Parte> partes) {...}
    public static Instrumento getInstrumento(
       String nome) {...}
}
public interface Instrumento {
    Collection<Parte> getParts();
}
```

Erro de Tipo

Exemplo 12. 14 – Código Genérico

```
package com.minhaEmpresa.inventario;
import br.com.MorteIndolor.coisas.*;
public class Lamina implements Parte { ...}
public class Guilhotina implements Parte { …}
public class Main {
  public static void main(String[] args) {
    Collection c = new ArrayList();
    c.add(new Guilhotina()) ;
    c.add(new Lamina());
    Inventario.addInstrumento(”Guilhotina”, c); // 1
    Collection k = Inventario.getInstrumento(
       ”Guilhotina”).getPartes();
  }
}
```

Exemplo 12. 15 – Usando Código Genérico em Código Legado

O código do cliente foi escrito antes da existência de generics, entretanto, ele usa o pacote da empresa Morte Indolor, e a biblioteca Collection, que foram implementados usando tipos parametrizados. Todas as definições do código cliente usam tipos crus.

A linha comentada 1 gera um alerta, pois uma Collection simples está sendo passada onde uma coleção de Parte é esperada, e o compilador não pode confirmar se essa atribuição é válida.

1.6 – Alguns Detalhes sobre Generics

Nesta seção são apresentados vários pormenores a respeito dos generics de Java.

1.6.1 – Compartilhamento de Classe Genérica

Pense no que será impresso pela rotina abaixo.

```
List <String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass());
```

A resposta é que será impresso "True". A justificativa é que ambas definições fazem uso da mesma classe em tempo de execução, mesmo que seus tipos reais sejam diferentes. Realmente, o que faz uma classe de Java ser genérica é que ela tem o mesmo comportamento para todos os parâmetros reais de tipo em que ela for instanciada. A mesma classe pode ser tratada como tendo vários tipos. Como conseqüência, as variáveis estáticas e os métodos estáticos são compartilhados por todas as instâncias.

1.6.2 – Casts e InstanceOf

Outra implicação de uma classe genérica ser compartilhada por todas as suas instâncias é que não faz sentido perguntar a uma instância se ela é de um tipo particular de invocação de um tipo genérico.

```
Collection cs = new ArrayList<String>();
if (cs instanceof Collection<String>) { ...} // ilegal
```

Da mesma forma, o cast seguinte é inútil:

```
Collection<String> cstr = (Collection<String>) cs;
// alerta de "não verificado"
```

O mesmo acontece com variáveis de tipo:

```
<T> T badCast(T t, Object o) {
    return (T) o; // alerta de "não verificado"
 }
```

Variáveis de tipo não existem durante a execução do programa. Isso quer dizer que não é requerido mais memória nem há overhead no desempenho, o que é bom. Por outro lado, infelizmente, conforme visto, não se pode confiar no uso de casts a partir delas.