nemo

ontology & conceptual modeling research group

Desenvolvimento OO com Java

12 – Tipos Genéricos

Vítor E. Silva Souza

(vitorsouza@inf.ufes.br)

http://www.inf.ufes.br/~vitorsouza

Departamento de Informática

Centro Tecnológico

Universidade Federal do Espírito Santo

Objetivos da parte 12

- Mostrar como tipos genéricos melhoram a redigibilidade, legibilidade e confiabilidade;
- Demonstrar como usar e construir tipos genéricos;
- Apresentar brevemente detalhes sobre esta novidade do Java 5.0.

Tipos genéricos

- Novidade do Java 5.0;
- Funcionalidade já existente em outras linguagens (ex.: templates do C++);
- Teoria estudada e solidificada;
- Muitas bibliotecas são genéricas:
- Código complicado de ler e manter;
- Coerção leva a erros em tempo de execução.

Tipos genéricos e coleções

```
// Java 1.4:
Lista lista = new ArrayList();
lista.add(new Integer(100));
int numero = ((Integer)lista.get(0)).intValue();

// Com tipos genéricos:
Lista<Integer> lista = new ArrayList<Integer>();
lista.add(new Integer(100));
int numero = lista.get(0).intValue();

// Com tipos genéricos e autoboxing:
Lista<Integer> lista = new ArrayList<Integer>();
lista.add(100);
int numero = lista.get(0);
```

Tipos genéricos e Comparable

```
// Java 1.4:
class Pessoa implements Comparable {
  private String nome;
  public int compareTo(Object o) {
     Pessoa p = (Pessoa)o;
     return nome.compareTo(p.nome);
  }
}

// Com generics:
class Pessoa implements Comparable<Pessoa> {
  private String nome;
  public int compareTo(Pessoa o) {
     return nome.compareTo(o.nome);
  }
}
```

Funcionamento

- Cria-se uma classe “genérica”:
  - Trabalha com um tipo T, desconhecido;
  - Tipo será atribuído na definição da referência.

```
public class Casulo<T> {
  private T elemento;
  public void colocar(T elem) {
     elemento = elem;
  }
  public T retirar() {
     return elemento;
  }
}
```

Funcionamento

- Referência e construtor definem o tipo manipulado pela classe genérica;
- Compilador pode efetuar checagens de tipo.

```
Casulo<String> cs = new Casulo<String>();
cs.colocar("Uma string");
// Erro: cs.colocar(new Integer(10));
String s = cs.retirar();

Casulo<Object> co = new Casulo<Object>();
co.colocar("Uma string");
co.colocar(new Integer(10));
Object o = co.retirar();
```

Herança de tipos genéricos

- Os conceitos de herança podem se confundir quando usamos tipos genéricos.

```
Casulo<String> cs = new Casulo<String>();
Casulo<Object> co = cs;
```

- O código acima gera erro;
- Por que? `Object` não é superclasse de `String`?

```
co.colocar(new Integer());  // OK!
String s = cs.retirar();    // OK!
```

- co e cs são o mesmo objeto e o código acima faria s receber um `Integer`!

Coringas (*wildcards*)

- Considere, então, um código genérico:

```
void imprimir(Casulo<Object> c) {
  System.out.println(c.retirar());
}
```

- O código não é tão genérico assim:

```
imprimir(co);   // OK!
imprimir(cs);   // Erro!
```

- Como acabamos de ver, não se pode converter `Casulo<String>` para `Casulo<Object>`!

Coringas (*wildcards*)

- Para essa situação, podemos usar coringas:

```
void imprimir(Casulo<?> c) {
  System.out.println(c.retirar());
}
```

- Significa: o método `imprimir()` pode receber casulos de qualquer tipo.

Coringas limitados

- Podemos também limitar o tipo genérico como sendo subclasse de alguma classe;

```
void desenhar(Casulo<? extends Forma> c) {
  c.retirar().desenhar();
  c.retirar().inverter();
}
```

- Significa: o método `imprimir()` pode receber casulos de qualquer subclasse de `Forma` ou casulos de `Forma`;
- O compilador garante que o que retirarmos do casulo será uma `Forma` ou uma subclasse.

Coringas limitados

- Porém, não podemos alterar uma classe com um coringa:

```
void teste(Casulo<? extends Forma> c) {
  // Erro: c pode ser Casulo<Retangulo>!
  c.colocar(new Circulo());
}

void outroTeste(Casulo<?> c) {
  // Erro: c pode ser Casulo<Integer>!
  c.colocar("Uma string!");
}

void desenhar(Casulo<? extends Forma> c) {
  c.retirar().desenhar();
  c.retirar().inverter();
}
```

Métodos genéricos

- Novamente, este método não é tão genérico:

```
static void arrayToCollection(Object[] a,
                              Collection<Object> c) {
    for (Object o : a) c.add(o);
}
```

- E este gera erro de compilação:

```
static void arrayToCollection(Object[] a,
                              Collection<?> c) {
    for (Object o : a) c.add(o); // Erro!
}
```

Métodos genéricos

- A solução:

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

- O método usa um tipo diferente a cada chamada:

```
// usa Long[] e Collection<Long>:
arrayToCollection(new Long[] {1L, 2L},
                  new ArrayList<Long>());
```

Código legado

- Ao misturar código novo com código pré-genéricos, podem ocorrer problemas:

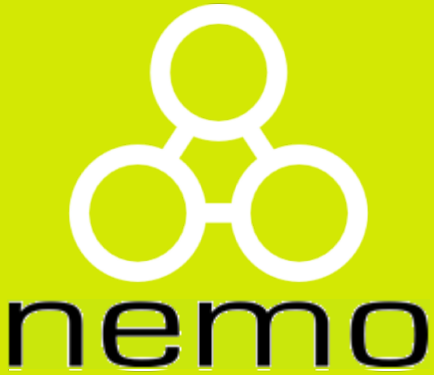
```
public class Teste {
  static Collection listaInt() {
    List l = new ArrayList();
    l.add(1); l.add(2); l.add(3);
    return l;
  }

  public static void main(String[] args) {
    Collection<Integer> lista = listaInt();
  }
}
```

Código legado

- Ao contrário do que se pode imaginar, `Collection` ≠ `Collection<Object>`;
- Classes sem definição do tipo genérico podem ser convertidas para classes com definição de qualquer tipo;
- Gera um aviso (*warning*) e o desenvolvedor deve assegurar que não haverá erro de execução.

Conclusões

- Usar tipos genéricos é relativamente simples e traz grandes vantagens;
- Criar tipos genéricos é mais complexo e envolve um entendimento mais aprofundado.



http://nemo.inf.ufes.br/