Computação II – Orientação a Objetos

Fabio Mascarenhas - 2014.1

http://www.dcc.ufrj.br/~fabiom/java

Classes parametrizadas (*classes genéricas*)

- Várias classes da biblioteca padrão de Java têm *parâmetros*
- Por exemplo, todas as classes que representam coleções (listas, mapas, conjuntos) recebem um parâmetro que diz quais são os objetos que fazem parte da coleção
- Os parâmetros são outras classes, e aparecem entre <>
  - `Set<String>`, `ArrayList<Tijolo>`, `HashMap<String,Aluno>`
- Depois veremos como definir nossas próprias classes parametrizadas

Listas de Strings

- Vamos supor que temos uma interface para representar listas simples de strings:

```
public interface Lista {
    int tamanho();
    String le(int i);
    void escreve(int i, String s);
    void adiciona(String s);
}
```

- Uma implementação possível seria uma lista usando um vetor que cresce, outra uma lista usando uma lista ligada

- Mas todas são listas de strings!

Listas de Objetos

- Se quisermos uma listas de Figura, precisamos de outra interface, e outras implementações; para listas de inteiros, mais outra interface, e outras implementações... como eliminar essa duplicação?

- Podemos pensar em uma lista de *objetos*:

```
public interface Lista {
    int tamanho();
    Object le(int i);
    void escreve(int i, Object s);
    void adiciona(Object s);
}
```

Listas de Objetos

- Agora podemos representar listas de qualquer coisa, mas com dois inconvenientes

- Quando lemos um objeto da lista, precisamos de uma conversão de tipo para poder ter de volta o objeto do jeito que o botamos lá:

```
l.adiciona("foo");
String s = (String)l.le(0);
```

- E o compilador não controla o que pode ser adicionado na lista; um erro de programação pode por um inteiro em nossa “lista de strings”, ou uma string em nossa lista de figuras, e só vamos saber na hora de executar o programa

Tipos Parametrizados ou Genéricos

- Os tipos parametrizados ou genéricos são a solução para esse problema

- Podemos *parametrizar* nossa lista pelo *tipo do elemento*, declarando um *parâmetro de tipo* na sua definição:

TIPO ELEMENTO

```
public interface Lista<E> {
    int tamanho();
    E le(int i);
    void escreve(int i, E s);
    void adiciona(E s);
}
```

- `E` é o parâmetro de tipo que indica o tipo dos elementos da lista; por convenção, usamos nomes curtos e em maiúsculas, mas poderia ser qualquer nome

Usando Parâmetros

- Um parâmetro de tipo indica um tipo, logo pode ser usado em qualquer lugar onde usamos o nome de uma classe, ou de uma interface: tipos de campos, tipos de parâmetros de métodos, tipos de variáveis...

- Só não podemos usá-lo diretamente com `new`: `new E()` é um erro de compilação

- Podemos usá-lo em um tipo vetor, mas não podemos instanciar esse vetor diretamente: `new E[10]` é um erro de compilação

- Podemos usá-lo também como parâmetro em outros tipos parametrizados, como veremos a seguir

Implementando Lista<E>

- Uma maneira simples de implementa Lista<E> é com a classe a seguir:

```
public class ListaVetor<E> implements Lista<E> {
    private ArrayList<E> al = new ArrayList<E>();
    public int tamanho() {
        return al.size();
    }
    public E le(int i) {
        return al.get(i);
    }
    public void escreve(int i, E s) {
        al.set(i, s);
    }
    public void adiciona(E s) {
        al.add(s);
    }
}
```

Múltiplos Parâmetros

- Tipos parametrizados podem ter vários parâmetros de tipos, como a interface abaixo, que representa funções de um parâmetro:

```
public interface Funcao<E,S>
{
    S aplica(E x);
}
```

- Uma classe que implementa `Funcao` não precisa ela própria ser parametrizada:

```
public class Tamanho implements Funcao<String,Integer>
{
    public Integer aplica(String s) {
        return s.length();
    }
}
```

Subtipagem

- Duas instâncias do mesmo tipo paramétrico não são subtipo uma da outra se seus parâmetros são diferentes, mesmo que um parâmetro seja subtipo do outro

- `Lista<String>` **não é** subtipo de `Lista<Object>`!

- É fácil ver por quê:

```
Lista<String> ls = new ListaVetor<String>();
Lista<Object> lo = ls; // ok se Lista<String> é
                       // subtipo de Lista<Object>
lo.add(10); // ok pois 10 é um Object
String s = ls.le(0); // oops...
```

Coringas

- Nunca ter subtipagem entre tipos parametrizados é muito restritivo! É seguro tratar uma `Lista<String>` como uma `Lista<Object>`, ou uma `Lista<Retangulo>` como uma `Lista<Figura>`, contanto que sempre *retiremos* objetos da lista, nunca *coloquemos* objetos nela

- Para isso podemos usar um *coringa* `?` Como parâmetro de tipo:

```
Lista<String> ls = new ListaVetor<String>();
ls.add("foo");
Lista<?> lo = ls;
Object o = lo.le(0); // o tipo do coringa é Object
lo.add(10); // erro de compilação
```

- Se usamos um coringa, não podemos chamar nenhum método que use o parâmetro na assinatura, mas podemos chamar métodos que usem o parâmetro no tipo de retorno, e métodos que não usem o parâmetro

Coringas com limite

- Usando um coringa, podemos ter uma `Lista<Object>` que pode apontar para qualquer lista de maneira segura, mas e se quisermos um tipo mais específico, como uma Lista<Figura> que pode apontar para uma Lista<Retangulo>, ou Lista<Circulo>?

- Podemos definir um *limite* para o coringa:

```
Lista<Retangulo> ls = new ListaVetor<Retangulo>();
ls.add(new Retangulo(2, 3, 50, 100));
Lista<? extends Figura> lo = ls;
Figura o = lo.le(0); // o tipo do coringa é Figura
```

Limite inferior

- Em que tipos de lista podemos adicionar um `Retangulo`? `Lista<Retangulo>`, `Lista<FiguraPt>`, `Lista<Figura>`, Lista<Object>

- Podemos representar essas listas com um coringa com um *limite inferior*:

```
Lista<? super Retangulo> ls = new ListaVetor<Figura>();
ls.adiciona(new Retangulo(2, 3, 50, 100));
Object o = ls.le(0); // o tipo do coringa é Object
ls.adiciona(new Circulo(2, 3, 50)); // erro de compilação
```

- E se quisermos ter métodos `copiaDe` e `copiaPara` em `Lista<E>`, para adicionar todos os elementos da lista passada na atual, e para adicionar todos os elementos da lista atual na lista passada?

Métodos Genéricos

- Coringas não ser suficientes para escrever métodos genéricos; vamos pensar em um método coleta em `Lista<E>`, que recebe uma `Funcao` e retorna nova lista que é o resultado de passar todos os elementos da lista atual por essa `Funcao`:

```
Lista<?> coleta(Funcao<?,E> f);
```

- No final perdemos o tipo que a função está produzindo! Precisamos de um parâmetro para ele, mas se usar `E` é muito restritivo, então podemos dar um parâmetro pro método:

```
<S> Lista<S> coleta(Funcao<E,S> f);
```