Computação II – Orientação a Objetos

Fabio Mascarenhas - 2016.2

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

Subtipagem e Coringas

- `Lista<T>` é subtipo de `Lista<?>` para qualquer *T*
  - Não podemos chamar métodos em `Lista<?>` que recebem `?`, e `?` tem tipo `Object` quando chamamos métodos que retornam `?`
- `Lista<T1>` é subtipo de `Lista<? extends T2>` se *T1* é subtipo de *T2*
  - Como acima, mas `?` tem tipo *T2* quando um método retorna `?`
- `Lista<T1>` é subtipo de `Lista<? super T2>` se *T2* é subtipo de *T1*
  - Podemos chamar métodos que recebem `?` com subtipos de *T2*, e `?` tem tipo `Object` quando chamamos métodos que retornam `?`

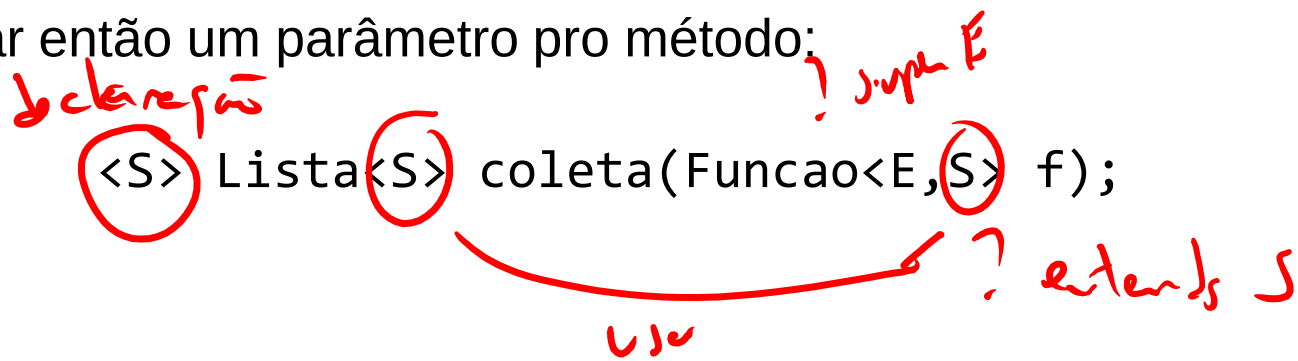
Métodos Genéricos

- 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, e usar `E` seria muito restritivo

- Podemos dar então um parâmetro pro método:



Inferência de parâmetros

- A chamada a um método genérico geralmente é igual à chamada de um método normal

- Os parâmetros de tipo do método são *inferidos* a partir do contexto da chamada

- Se não estivermos satisfeitos com os tipos inferidos podemos especificá-los explicitamente:

```
lista.<Object>coleta(f)
```

- Java 8 também podem inferir parâmetros na *instantiação* de classes genéricas, basta deixar a lista de argumentos de tipo vazia:

```
Lista<String> ls = new ListaVetor<>();
```

Limites nos Parâmetros

- Não são apenas coringas que podem ter um limite com `extends`, parâmetros de tipo normais também podem ter

- Um limite permite chamar os métodos do tipo que damos como limite, e passar uma referência do tipo do parâmetro para métodos que esperam o tipo do limite:

```
public class ListaFigura<E extends Figura> extends ListaVetor<E> {
    public void desenhar() {
        for(int i = 0; i < tamanho(); i++) {
            le(i).desenhar();
        }
    }
}
```

Sutilezas dos Tipos Genéricos

- Tipos genéricos só existem em tempo de compilação, não fazem parte da implementação de Java
- Em geral, parâmetros são convertidos para `Object` pelo compilador, a não ser que ele tenha um limite com `extends`
- O compilador também introduz conversões de tipo, e o código final é parecido com o da “lista de objetos”
- Essa “deleção” dos tipos parametrizados tem diversas consequências que estão descritas na documentação

Laço `for` de coleções

- Java tem uma versão do laço for que é especializada para percorrer uma coleção (um vetor, ou instâncias de uma das classes de coleção como `HashSet` e `ArrayList`)

- O bloco do laço é executado para cada elemento da coleção, com a variável de controle apontando para o elemento

```
for(int i: vetor) {
   System.out.println(i);
}
```

Iterable<E> e Iterator<E>

- Qualquer classe pode ser usada com um for de coleções, contanto que implemente a interface parametrizada Iterable<E>:

```
public interface Iterable<E> {
   Iterator<E> iterator();
}
```

- A interface `Iterator<E>` é uma versão genérica da nossa `Enumerador`:

```
public interface Iterator<E> {
    boolean hasNext();
    E next(); // lança NoSuchElementException
    void remove();
}
```

Comparable<T>

- Uma interface para valores que “ordenáveis”

```
interface Comparable<T> {
    int compareTo(T outro);
}
```

- A classe `String` implementa a interface `Comparable<String>`

- O parâmetro de tipo serve para restringir o que pode ser passado para `compareTo`, para não ter o mesmo problema de `equals`!

- Note a assinatura da função `Collections.sort`:

```
public static <T extends Comparable<? super T>>
                                    void sort(List<T> list);
```

- Prima da interface `Comparator<T>`, para um “ordenador” de `T`s (com um método `compare(T um, T outro)`)

java.util.function

- O pacote `java.util.function` de Java 8 adicionou várias interfaces genéricas de um método, para serem usadas com as referências a métodos e *expressões lambda* de Java 8

- Várias são versões genéricas de interfaces que já usamos: `Consumer<T>` é uma versão genérica de um `Comando` e `Acao`, `Function<T,R>` é a interface `Funcao<E,S>` que já vimos, `Predicate<T>` é uma interface genérica para representar expressões condicionais, `Supplier<T>` é uma versão genérica de interfaces como `Texto` que usamos em nosso toolkit gráfico

- A documentação da linguagem tem uma lista completa (https://goo.gl/DtU93N)

Expressões Lambda

- Além das referências a métodos, Java 8 introduziu mais uma forma de criar instâncias anônimas de interfaces com um único método abstrato: as *expressões lambda*

- Uma expressão lambda é uma lista de parâmetros e uma expressão ou um bloco: a lista de parâmetros tem o mesmo número de parâmetros do método que está sendo implementado implicitamente, e a expressão ou bloco é seu corpo

```
(params) -> exp          (params) -> { bloco }
```

- Tipos nos parâmetros são opcionais, pois o método que está sendo implementado já diz quais eles são

- Até os parênteses em volta dos parâmetros são opcionais se tiver um parâmetro só!