02

Escreva uma expressão Lambda simples que consuma uma expressão Lambda Predicate

Escreva uma expressão Lambda simples que consuma uma expressão Lambda Predicate

Entre as diversas novidades do Java 8, uma que se destaca bastante é a inclusão de ::lambdas::, trazendo algumas características funcionais à linguagem. Para ver melhor a utilidade dos ::lambdas:: e entender o que será cobrado na prova, vamos focar no exemplo a seguir.

Imagine que você precise escrever um método que recebe como parâmetros uma lista e um critério de busca, e retorna outra lista com os elementos que atendem ao critério. Poderíamos implementar este código da seguinte maneira:

```
class Person {
    private String name;
    private int age;
    //...
}

interface Matcher<T>{
    boolean test(T t);
}

class PersonFilter{

    public List<Person> filter(List<Person> input,
                               Matcher<Person> matcher){
```

```
        List<Person> output = new ArrayList<>();
        for (Person person : input) {
            if(matcher.test(person)){
                output.add(person);
            }
        }
        return output;
    }
}
```

COPIAR CÓDIGO

Ótimo, nossa base está montada. Agora, para filtrar de uma lista de pessoas apenas as maiores de idade, podemos implementar um `Matcher` da seguinte maneira:

```
class AgeOfMajority implements Matcher<Person>{
    @Override
    public boolean test(Person p) {
        return p.getAge() >= 18;
    }
}
```

COPIAR CÓDIGO

E usar esta classe em nosso código:

```
PersonFilter pf = new PersonFilter();
List<Person> adults = pf.filter(persons, new AgeOfMajority());
```

COPIAR CÓDIGO

O problema dessa abordagem é que, sempre que quisermos um critério diferente, precisamos criar uma nova classe que implemente `Matcher`, mesmo se for para usar apenas uma vez. Podemos reduzir um pouco esse impacto usando classes anônimas, mas a legibilidade do código fica prejudicada:

```
List<Person> adults = pf.filter(persons, new Matcher<Person>()
{
    @Override
    public boolean test(Person p) {
        return p.getAge() >= 18;
    }
});
```

COPIAR CÓDIGO

É para resolver este tipo de problema que existem os ::lambdas::. Um ::lambda:: é um trecho de código que pode ser passado como parâmetro para um método ou ser armazenado em uma variável para ser invocado posteriormente.

Para usar um ::lambda:: em Java, precisamos de uma **interface funcional**. Interfaces funcionais são interfaces normais, mas com apenas um método. Nossa interface `Matcher` pode ser considerada funcional. É possível checar se uma interface é funcional usando a ::annotation:: `FunctionalInterface`, se não for funcional, o código não compila:

```
@FunctionalInterface
interface Matcher<T>{
    boolean test(T t);
}
```

COPIAR CÓDIGO

O Java já vem com várias dessas interfaces funcionais para os cenários comuns. Uma destas interfaces é a `Predicate`, que recebe um objeto como parâmetro e retorna um `boolean`, exatamente igual a nosso `Matcher`. Vamos trocar o código da classe `PersonFilter` para usar o `Predicate`:

```
import java.util.function.Predicate;

class PersonFilter{

    public List<Person> filter(List<Person> input,
                               Predicate<Person> matcher){
        List<Person> output = new ArrayList<>();
        for (Person person : input) {
            if(matcher.test(person)){
                output.add(person);
            }
        }
        return output;
    }
}
```

COPIAR CÓDIGO

E para fazer o filtro:

```
Predicate<Person> matcher = new Predicate<Person>() {
    @Override
    public boolean test(Person p) {
        return p.getAge() >= 18;
    }
};
List<Person> adults = pf.filter(persons, matcher);
```

COPIAR CÓDIGO

Ok, vamos converter este código para um ::lambda::. A sintaxe básica do lambda é a seguinte:

```
( parameters ) -> { code }
```

COPIAR CÓDIGO

Usando esta fórmula no código anterior temos o seguinte:

```
Predicate<Person> matcher =
    (Person p) -> {return p.getAge() >= 18;};
```

COPIAR CÓDIGO

Perceba quanto código foi removido. Praticamente toda a declaração do tipo, de que não precisamos, já que a declaração do tipo da variável tem o mesmo de forma explícita. Também removemos o nome do método, que também não é necessário, já que interfaces funcionais possuem apenas um método.

Podemos remover mais código ainda. Repare que a variável `matcher` é do tipo `Predicate<Person>`. Aqui podemos inferir o tipo do parâmetro pelo tipo ::generics:: da interface. o código fica:

```
Predicate<Person> matcher = (p) -> {return p.getAge() >= 18;};
```

COPIAR CÓDIGO

Se temos apenas um argumento, podemos ainda remover os parênteses:

```
Predicate<Person> matcher = p -> {return p.getAge() >= 18;};
```

COPIAR CÓDIGO

Se temos apenas uma linha de código dentro do ::lambda::, podemos omitir as chaves. Se esta linha for o retorno, podemos omitir a palavra `return` também:

```
Predicate<Person> matcher = p -> p.getAge() >= 18;
```

COPIAR CÓDIGO

Pronto, já retiramos bastante código, está bem mais limpo. Nosso código no final fica assim:

```
Predicate<Person> matcher = p -> p.getAge() >= 18;
List<Person> adults = pf.filter(persons, matcher);
```

COPIAR CÓDIGO

Não somos obrigados a armazenar o ::lambda:: em uma variável, podemos passá-lo diretamente como parâmetro do método:

```
List<Person> adults = pf.filter(persons, p -> p.getAge() >=
18);
```

COPIAR CÓDIGO

Vamos entender qual a vantagem desta abordagem. Se agora precisarmos de um outro filtro, que retorna apenas as pessoas cujo nome comece com a letra "A", podemos simplesmente fazer:

```
List<Person> namesStartingWithA =
    pf.filter(persons, p -> p.getName().startsWith("A"));
```

COPIAR CÓDIGO

Não há necessidade de criar classes, nem mesmo anônimas. A inclusão dos ::lambdas:: nos permite escrever código altamente adaptável e ainda reduzir muito a verbosidade comum do Java.

Antes de passar para o próximo exemplo, vamos entender as regras para se escrever um ::lambda::.

- ::Lambdas:: podem ter vários argumentos, como um método. Basta separá-los por `,`.
- O tipo dos parâmetros pode ser inferido e, assim, omitido da declaração.
- Se não houver nenhum parâmetro, é necessário incluir parênteses vazios, como em:

```
Runnable r = () -> System.out.println("a runnable object!");
```

COPIAR CÓDIGO

- Se houver apenas um parâmetro, podemos omitir os parênteses, como em:

```
Predicate<Person> matcher = p -> p.getAge() >= 18;
```

COPIAR CÓDIGO

- O corpo do ::lambda:: pode conter várias instruções, assim como um método.
- Se houver apenas uma instrução, podemos omitir as chaves, como em :

```
Predicate<Person> matcher = p -> p.getAge() >= 18;
```

COPIAR CÓDIGO

- Se houver mais de uma instrução, é necessário delimitar o corpo do ::lambda:: com chaves, como em:

```
Runnable r = () ->  {
    int a = 10;
    int b = 20;
    System.out.println(a + b);
}
```

COPIAR CÓDIGO

Acessando variáveis do objeto com ::lambdas::

::Lambdas:: podem interagir com as variáveis de instância dos objetos onde foram declarados. Temos apenas que tomar cuidado com variáveis marcadas como `final` :

```
public class LambdaScopeTest {

    public int instanceVar = 1;
    public final int instanceVarFinal = 2;

    public static void main(String[] args) {
        new LambdaScopeTest().test();
    }

    private void test() {
        instanceVar++; // ok
        new Thread(() -> {
            System.out.println(instanceVar); // ok
            instanceVar++; // ok

            System.out.println(instanceVarFinal); // ok
            instanceVarFinal++; // compile error
        }).start();
    }
```

COPIAR CÓDIGO

Já com variáveis locais de método e parâmetros, as regras são um pouco mais complexas. ::Lambdas:: só podem interagir com variáveis locais caso estas estejam marcadas como `final` (uma referência imutável) ou que sejam **efetivamente final** (não são `final`, mas não são alteradas). Não é possível alterar o valor de nenhuma variável local dentro de um ::lambda:::

```
    private void test() {
        int unchangedLocalVar = 3;   // effectively final
        final int localVarFinal = 4; // final
        int simpleLocalVar = 0;
```

```
        simpleLocalVar = 9; // updated the value

        new Thread(() -> {
            System.out.println(unchangedLocalVar);   // can
read
            System.out.println(localVarFinal);       // can
read
            System.out.println(simpleLocalVar); // compile
error
        }).start();
    }
```

COPIAR CÓDIGO

Conflitos de nomes com ::lambdas::

As variáveis do ::lambda:: são do mesmo escopo que o método onde ele foi declarado, portanto, não podemos declarar nenhuma variável, como parâmetro ou dentro do corpo, cujo nome conflite com alguma variável local do método:

```
private void test(String param) {
    String methodVar = "method"; //not final

    Predicate<String> a =
        param -> param.length() > 0; //compile error
    Predicate<String> b =
        methodVar -> methodVar.length() > 0; //compile error
    Predicate<String> c =
        newVar -> newVar.length() > 0; // ok
}
```

COPIAR CÓDIGO