01

Funções de collections e expressão lambda

Transcrição

Continuando com o processo de refatoração, voltaremos nossa atenção para a chamada do `ResumoView()` na *ListaTransacoesActivity*.

Repare que dentro da função `configuraResumo()`, estamos criando o objeto `resumoView`, e por meio dele, responsabilizamos a *activity* em chamar o `adicionaReceita()`, `adicionaDespesa()` e o `adicionaTotal()`.

Caso a *activity* se "esqueça" de chamar o `adicionaDespesa()` por exemplo, a informação não aparecerá na View. Essa estratégia pode ser muito perigosa. Então, como podemos melhorar esse aspecto?

Podemos criar uma única função dentro do `resumoView` que irá conter todas as funções (`adicionaReceita()`, `adicionaDespesa()`, `adicionaTotal()`). Sendo assim, a *activity* só chamará apenas um único método que contém todos os outros, e ela não terá que se responsabilizar em chamar todas as funções, e correr o risco de esquecer alguma nova função posteriormente.

Acessaremos o `ResumoView()`, e criaremos uma nova função chamada `atualiza()`. Essa função será responsável por chamar todas as funções citadas acima *internamente*.

```
fun atualiza() {
    adicionaReceita()
    adicionaDespesa()
    adicionaTotal()
}
```

Com essa refatoração, as três funções que contém cada uma a sua lógica, não precisam mais estarem visíveis para o público, e nem para quem for utilizar a `ResumoView()`.

Quer for usar a `ResumoView()` só precisa saber que existe o método `atualiza()`, que chama internamente todos os outros métodos. Por isso, podemos adicionar o modificador **private** nesses métodos.

```
private fun adicionaReceita() {
    // conteúdo do método
}

private fun adicionaDespesa() {
    // conteúdo do método
}

private fun adicionaTotal() {
    // conteúdo do método
}
```

Dessa maneira, não deixamos que alguém se responsabilize em chamar esses métodos, somente o `ResumoView()`.

Na *activity* `ListaTransacoesActivity`, há código quebrado dentro da função `configuraResumo()`. Isso acontece porque ela não consegue mais enxergar as função, pois as mesmas estão *privadas*. A única função que a *activity* irá enxergar é a

`atualiza()`.

```
private fun configuraResumo(transacoes: List<Transacao>) {
    val view: View = window.decorView
    val resumoView = ResumoView(context: this, view, transacoes)
    resumoView.atualiza()
}
```

Vamos testar.

O Android Studio conseguiu executar, e o *app* ainda mantém o mesmo aspecto visual. Agora, daremos uma olhada na classe `Resumo()`.

Em `Resumo()`, recebemos uma lista de transações, e dentro dela temos a função `receita()`, `despesa()` e `total()`. São funções que realizam cálculos para nós.

Mas, para saber se realmente elas cumprem suas tarefas, olhe o quanto de código é necessário interpretarmos: em `receita()`, temos um laço para cada transação, pegamos todas as transações de *receita* para depois fazer o cálculo.

Será que existe uma maneira mais objetiva de realizar todo esse cálculo utilizando o **Kotlin**?

Felizmente, como o Kotlin é *multiparadigma*, tanto orientado a objetos quanto funcional, ele possui alguns recursos que nos auxiliam na manipulação de **collections**. Com isso, conseguimos fazer uma manipulação muito mais objetiva, do que fazer filtros com `if()`, `for` e outras maneiras.

Veremos agora, como podemos resumir mais esse código, mantendo o mesmo resultado.

Olharemos da seguinte maneira: podemos ver que estamos passando por cada uma das transações no `for()`, e no `if()`, realizamos um filtro de todas as transações que são do tipo **receita**.

Seria bem mais simples se a lista `transacoes` pudesse listar utilizando o método `filter()`, que existe no Kotlin. Essa função permite que nós selecionemos e filtremos itens de uma lista, ela espera uma *implementação* de uma função dentro de si.

Essas funções implementadas são conhecidas como **lambda** ou **funções anônimas**. Para implementar essas funções precisamos "abrir e fechar chaves":

```
transacoes.filter({})
```

A partir do momento em que "abrimos e fechamos chaves", estamos declarando uma **lambda**. Para conseguir pegar um item dessa função anônima, primeiro é necessário colocar um apelido nesse item, que será `transacao`. No momento em que estamos nomeando esse item, aparece uma sugestão de nome com uma flecha.

```
transacao ->
```

O segundo passo para fazer a operação `filter()` é a operação booleana. Então, copiaremos o teste booleano do `if()`, e colocaremos logo após a seta.

```
class Resumo(private val transacoes: List<Transacao>) {

    fun receita() : BigDecimal{
        var totalReceita = BigDecimal.ZERO
        for (transacao in transacoes) {
            if (transacao.tipo == Tipo.RECEITA) {
                totalReceita = totalReceita.plus(transacao.valor)
            }
        }

        transacoes.filter({transacao -> transacao.tipo == Tipo.RECEITA })

        return totalReceita
    }

    fun despesa() : BigDecimal {...}

    fun total() : BigDecimal {...}
}
```

Repare que no momento em que estamos chamando o filtro, estamos mandando uma *lambda*, e estamos nomeando de `transacao` cada item dentro da lista de transações. E a operação em questão é pegar o tipo da transação e verificar se ele é igual ao tipo `receita`.

Assim como no `if()`, vamos somar utilizando a função do Kotlin `sum()`. O Kotlin nos fornece duas variaçoes de soma, sendo elas `sumBy()` para somar valores inteiros e `sumByDouble()` para somar valores que tem ponto flutuante. Já que estamos lidando com ponto flutuante, utilizaremos `sumByDouble()`.

Da mesma forma que fizemos no `filter({})`, também abriremos o escopo para a *expressão lambda*.

```
class Resumo(private val transacoes: List<Transacao>) {

    fun receita() : BigDecimal{
        var totalReceita = BigDecimal.ZERO
        for (transacao in transacoes) {
            if (transacao.tipo == Tipo.RECEITA) {
                totalReceita = totalReceita.plus(transacao.valor)
            }
        }

        transacoes
            .filter({transacao -> transacao.tipo == Tipo.RECEITA})
            .sumByDouble({transacao -> })

        return totalReceita
    }

    fun despesa() : BigDecimal {...}

    fun total() : BigDecimal {...}
}
```

No geral, fizemos uma soma das transações que foram filtradas. Apelidamos o item que queríamos somar de `transacao ->`.

Agora, é necessário pegar o **valor** da transação. Mas repare que a soma `sumByDouble()` espera um **Double**, entretanto a `transacao.valor` devolverá um **BigDecimal**. Por isso é necessário converter este *valor* para **Double**, e conseguimos fazer isso facilmente, utilizando a função `toDouble()`.

```
transacoes
    .filter({transacao -> transacao.tipo == Tipo.RECEITA})
    .sumByDouble({transacao -> transacao.valor.toDouble()})
```

Com isso, conseguimos uma forma muito mais simples do que interpretar todo o código da função `receita()`.

Feito esse processo, vamos devolver o valor em uma variável chamada de `somaDeReceita` com o tipo `Double`:

```
var somaDeReceita: Double = transacoes
    .filter({ transacao -> transacao.tipo == Tipo.RECEITA })
    .sumByDouble({ transacao -> transacao.valor.toDouble() })

return totalReceita
```

Para poder reutilizar a soma de receita que acabamos de criar, primeiro é necessário comentar todo o código mais verboso anterior dentro de `receita()` utilizando o atalho "Ctrl + /":

```
fun receita() : BigDecimal{
//        var totalReceita = BigDecimal.ZERO
//        for (transacao in transacoes) {
//            if (transacao.tipo == Tipo.RECEITA) {
//                totalReceita = totalReceita.plus(transacao.valor)
//            }
//        }

        var somaDeReceita: Double = transacoes
            .filter({ transacao -> transacao.tipo == Tipo.RECEITA })
            .sumByDouble({ transacao -> transacao.valor.toDouble() })

        return totalReceita
    }
```

E depois, retornaremos um **BigDecimal** baseando-se na `somaDeReceita`.

```
        return BigDecimal(somaDeReceita)
```

Ao executar a aplicação, vimos que a app ainda está mantendo as informações mesmo com essa refatoração. Veja que o nosso código ficou muito mais simples de ler o que está acontecendo ao invés de ter que interpretar `for()` e `if()`. Agora podemos apagar o código comentado, pois não precisamos mais dele.

Já que estamos usando a *expressão lambda*, temos uma forma mais resumida de fazer a chamada da função dentro de outra função. Quando mandamos uma **única função** via parâmetro, temos a capacidade de não utilizar os parênteses,

deixando de uma forma mais objetiva.

```
fun receita() : BigDecimal{
    var somaDeReceita: Double = transacoes
        .filter { transacao -> transacao.tipo == Tipo.RECEITA }
        .sumByDouble { transacao -> transacao.valor.toDouble() }
    return totalReceita
}
```

O nosso código ficou muito mais simples de ser interpretado, além de que conseguimos utilizar a parte de expressão lambda e também, as funções anônimas.