05

Utilizando Single-Expression Function

Transcrição

Continuando com o processo de refatoração na classe `Resumo()`, repare a função `receita()` está bem mais simples agora, pois estamos utilizando o ***filter*** e o ***sumByDouble*** que facilita o entendimento do código.

Já a função `despesa()`, ainda está fazendo aquele caminho muito maior para somente realizar uma soma. Pensando em melhorar o aspecto desse código, pegaremos o código feito para a função `receita()` e colaremos dentro de `despesa()`. Assim, adaptaremos o código para que o cálculo seja feito para a despesa.

O primeiro passo é modificar a variável `somaDeReceita` para `somaDeDespesa` na função `despesa()`, utilizando o atalho "Shift + F6".

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

fun despesa() : BigDecimal {
    val somaDeDespesa: Double = transacoes
        .filter { transacao -> trasacao.tipo == Tipo.RECEITA }
        .sumByDouble { transacao -> transacao.valor.toDouble() }
    return BigDecimal(somaDeDespesa)
}
```

E, ao invés de realizarmos um filtro de `RECEITA`, faremos um de `DESPESA`.

```
fun despesa() : BigDecimal {
    val somaDeDespesa: Double = transacoes
        .filter { transacao -> trasacao.tipo == Tipo.DESPESA }
        .sumByDouble { transacao -> transacao.valor.toDouble() }
    return BigDecimal(somaDeDespesa)
}
```

Vamos executar o código com o atalho "Alt + Shift + F10". Legal, o resultado se manteve.

Como você pode ver, estamos utilizando o mesmo código em ambas as funções! E com isso, podemos *refatorar* o código, fazendo com que uma única função mantenha todo o código das transações. Assim, não será mais necessário repetir o código. Chamaremos essa função que irá conter o código repetido, a partir de um parâmetro específico, uma variável do tipo `Tipo`:

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

```
}

fun despesa() : BigDecimal {
    val somaDeDespesa: Double = transacoes
        .filter { transacao -> trasacao.tipo == Tipo.DESPESA }
        .sumByDouble { transacao -> transacao.valor.toDouble() }
    return BigDecimal(somaDeDespesa)
}

fun somaPor(tipo: Tipo) {
    transacoes
        .filter { transacao -> trasacao.tipo == tipo }
        .sumByDouble { transacao -> transacao.valor.toDouble() }
}
```

Agora, os métodos `receita()` e `despesa()` somente chamarão o método `somaPor()` passando o tipo da transação:

```
fun receita() : BigDecimal {
    val somaDeReceita: Double = somaPor(Tipo.RECEITA)
    return BigDecimal(somaDeReceita)
}

fun despesa() : BigDecimal {
    val somaDeDespesa: Double = somaPor(Tipo.DESPESA)
    return BigDecimal(somaDeDespesa)
}

private fun somaPor(tipo: Tipo) : Double {
    return transacoes
        .filter { transacao -> trasacao.tipo == tipo }
        .sumByDouble { transacao -> transacao.valor.toDouble() }
}
```

Deixamos a nova função `somaPor()` com o modificador *private*, pois assim ela fica acessível apenas para os membros da classe `Resumo`.

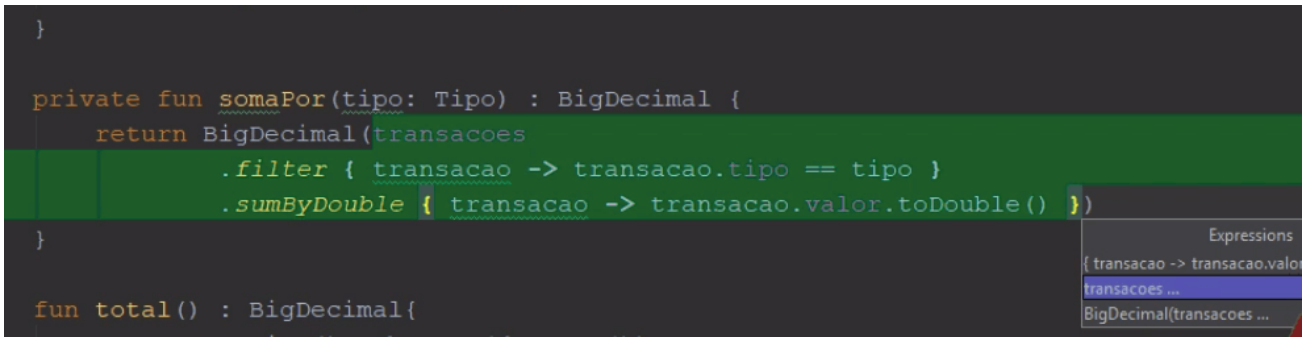
Vamos testar!

Muito bem. A refatoração manteve os dados do jeito que estavam antes.

Sobre o código, podemos deixá-lo ainda mais simples. Estamos devolvendo o `somaPor()` em uma variável - `somaDeReceita` ou `somaDeDespesa` - que está sendo retornada como um parâmetro para a instância de um BigDdecimal. Sendo assim, podemos retornar um BigDecimal na função `somaPor()`, pois é isso que já estamos esperando. Assim, nós também fazemos com que todo o código que realiza a soma, seja como um parâmetro de um BigDecimal:

```
private fun somaPor(tipo: Tipo) : BigDecimal {
    return BigDecimal(transacoes
        .filter { transacao -> trasacao.tipo == tipo }
        .sumByDouble { transacao -> transacao.valor.toDouble() })
}
```

Podemos também, extrair uma variável de toda a expressão que realiza as transações, colocando um `.val` no final de `{ transacao -> transacao.valor.toDouble() }`. Vamos escolher a opção "*transacoes...*":



Tudo isso será uma **somaDeTransacoesPeloTipo**:

```
private fun somaPor(tipo: Tipo) : BigDecimal {
    val somaDeTransacoesPeloTipo = transacoes
        .filter { transacao -> trasacao.tipo == tipo }
        .sumByDouble { transacao -> transacao.valor.toDouble() }
    return BigDecimal(somaDeTransacoesPeloTipo)
}
```

Legal. Ao invés de retornarmos para uma variável que nem é mais do tipo *Double*, ou até mesmo criar uma nova instância do *BigDecimal*, podemos simplesmente retornar apenas o `somaPor()`:

```
fun receita() : BigDecimal {
    return somaPor(Tipo.RECEITA)
}

fun despesa() : BigDecimal {
    return somaPor(Tipo.DESPESA)
}

private fun somaPor(tipo: Tipo) : BigDecimal {
    val somaDeTransacoesPeloTipo = transacoes
        .filter { transacao -> trasacao.tipo == tipo }
        .sumByDouble { transacao -> transacao.valor.toDouble() }
    return BigDecimal(somaDeTransacoesPeloTipo)
}
```

Bem mais simples, não é mesmo? Inclusive, há mais um detalhe na expressão lambda que falaremos agora.

Como podemos ver, colocamos apelidos para indicar cada item das transações. Entretanto, quando estamos utilizando essas *expressões lambdas*, já temos um objeto **subentendido** dentro dela, que indica todos os itens de uma coleção, ou seja, todo os itens da lista de transações.

E como podemos utilizar esse objeto subentendido?

Esse objeto é conhecido como **it**! E ele pode ser substituído pelo apelido e pela flecha.

```
private fun somaPor(tipo: Tipo) : BigDecimal {
    val somaDeTransacoesPeloTipo = transacoes
```

```
            .filter { it.tipo == tipo }
            .sumByDouble { it.valor.toDouble() }
        return BigDecimal(somaDeTransacoesPeloTipo)
    }
```

Esse é o objeto subentendido dentro de uma expressão lambda. E com isso, o nosso código ficou bem mais enxuto e simples de entender. Ao rodar novamente o programa, o resultado se manteve. O código ficou bem melhor, comparado ao que era anteriormente. Agora, tentaremos melhorar ainda mais o código.. será que isso é possível?

Além de tudo o que vimos aqui sobre lambda, sobre o paradigma funcional, sobre a manipulação de *collections*, temos um outro conceito de paradigma funcional, que é fazer com que o retorno das funções que tenham apenas uma **única linha**, seja feita em apenas **uma única linha**!

Essa é uma *feature* do Kotlin conhecida como **Single Expression Functions**, que consiste em dizer que a função tem como igualdade o `return`, fazendo com que o retorno seja *direto*, desta forma:

```
fun total() : BigDecimal = receita().subtract(despesa())
```

Com isso, não é mais necessário abrir um escopo para a função, e ser obrigados a colocar um `return`. Quando temos esse tipo de comportamento, também não precisamos deixar de forma explícita, o retorno da função. Vimos anteriormente que, todas as vezes que declaramos uma função, ou um retorno, tínhamos que colocar o `return` de forma explícita, pois não era claro, e então podiam ter outros tipos de retorno dentro da função.

Mas quando fazemos dessa maneira, utilizando *Single Expression Functions*, temos a capacidade de não colocar o retorno, porque já é claro qual seria o retorno dessa função.

Além da função `total()`, podemos fazer o mesmo com as funções `receita()` e `despesa()`, visto que ambas possuem somente uma linha de retorno.

```
fun receita() = somaPor(Tipo.RECEITA)

fun despesa() = somaPor(Tipo.DESPESA)

fun total() = receita().subtract(despesa())

private fun somaPor(tipo: Tipo) : Double {
    return transacoes
        .filter { transacao -> trasacao.tipo == tipo }
        .sumByDouble { transacao -> transacao.valor.toDouble() }
}
```

Vamos executar o código para ver o resultado. Tudo está funcionando como esperávamos, e ainda deixamos o código muito mais simples por utilizar a *Single Expression Functions*, permitindo que as funções retornem exatamente *expressões* da mesma maneira que atribuímos uma variável, por exemplo.

Além dessa forma, temos uma outra alternativa para deixar o código resumido. Com o cursor em cima da função `receita()`, e com o "Alt + Enter", temos a capacidade de converter uma função em uma *property* utilizando a opção *Convert function to property*. Ao selecionarmos essa opção, observe o que acontece:

```
val receita get() = somaPor(Tipo.RECEITA)

fun despesa() = somaPor(Tipo.DESPESA)

fun total() = receita().subtract(despesa())

private fun somaPor(tipo: Tipo) : Double {
    return transacoes
        .filter { transacao -> trasacao.tipo == tipo }
        .sumByDouble { transacao -> transacao.valor.toDouble() }
}
```

Essa opção fez com que a *property* seja uma `val` para não ser alterada, e ela faz com que o `get()` retorne a expressão `somaPor(Tipo.RECEITA)`. Assim temos o mesmo resultado da *Single Expression Functions*, mas agora no estilo de property. Faremos o mesmo em outros pontos do nosso código.

```
val receita get() = somaPor(Tipo.RECEITA)

val despesa get() = somaPor(Tipo.DESPESA)

val total get() = receita().subtract(despesa())
```

Quando alteramos as funções pra *properties*, o Android Studio conseguiu replicar a mudança em outros pontos do código, ou seja, a classe `ResumoView` está chamando `resumo.receita` via *property*, e não mais via função. Esse é um cuidado peculiar que devemos tomar, pois se fizéssemos a conversão na mão, seria preciso mudar em todos os outros pontos do código também.

Vamos executar a aplicação para ver se está tudo funcionando.

Como podemos ver, continuamos tendo o mesmo resultado após várias refatorações, e com um código muito mais simples.