05

Refatorando o código do resumo

Transcrição

Começaremos com o nosso processo de refatoração.

A princípio, podemos refatorar a parte em que pegamos a referência para a cor. Atualmente, pegamos essa referência várias vezes, e acabamos repetindo o código, tanto no `adicionaTotal()`, como no `adicionaDespesa()` e no `adicionaReceita()`.

Basicamente, podemos fazer com que a chamada dessas cores se tornem ***properties*** que representarão as cores de **receita** e de **despesa**.

Temos uma técnica que facilita o trabalho de modificar para uma *property*, e replicar para todos os pontos. Para utilizar essa técnica, é preciso selecionar todo o código que deseja extrair, e utilizar o atalho "Ctrl + Alt + F".

```
ContextCompat.getColor(context, R.color.receita)
```

Com esse atalho, vamos transformar o código selecionado em uma *property* da classe! Então, todos os pontos que utilizam exatamente o mesmo código, será modificado para a *property* que será criada. Precisamos que essa *property* fique dentro da classe `ResumoView`, pois de fato, é uma chamada dentro da classe. A outra opção que temos é a `ResumoView.kt` como se fosse deixá-la mais global, sem escopo de classe.

Após ter selecionado a opção `ResumoView`, obtemos esse resultado:

```
private val i = ContextCompat.getColor(context, R.color.receita)
```

Vamos trocar o nome `i` para `corReceita`. Assim, a função `adicionaReceita()` apresentará esse código:

```
fun adicionaReceita() {
    var totalReceita = resumo().receita()
    view.resumo_card_receita.setTextColor(corReceita)
    view.resumo_card_receita.text = totalReceita.formataParaBrasileiro()
}
```

Como vimos, todos os lugares que estavam utilizando a chamada da cor da receita, agora estão chamando a *property* `corReceita`. Podemos reparar que dentro do `adicionaTotal()`, a chamada para a cor da Receita também mudou.

Faremos o mesmo para chamada da cor da despesa. Selecionando o código dentro da função `adicionaDespesa()`, utilizarem ele será considerado a partir de um objeto que queremos executar.os o atalho "Ctrl + Alt + F". Após ter selecionado a classe `ResumoView`, trocaremos o nome da nova *property* de `i` para `corDespesa`. Da mesma forma, a função `adicionaDespesa()` sofrerá alterações.

```
fun adicionaDespesa() {
    var totalDespesa = resumo().despesa()
    view.resumo_card_despesa.setTextColor(corDespesa)
```

```
        view.resumo_card_despesa.text = totalDespesa.formataParaBrasileiro()
    }
```

Podemos isolar essas *properties* no começo da `ResumoView`:

```
class ResumoView(private val context:Context, private val view: View, transacoes: List<Transaca

    private val resumo: Resumo = Resumo(transacoes)
    private val corReceita = ContextCompat.getColor(context, R.color.receita)
    private val corDespesa = ContextCompat.getColor(context, R.color.despesa)
}
```

Assim, o nosso código ficou bem simples. Entretanto, repare que sempre que queremos setar uma cor ou colocar um valor nos componentes, estamos chamando `view.resumo_card_despesa` ou `view.resumo_card_receita`.

Será que não temos uma maneira mais objetiva de fazer essa chamada? Felizmente no Kotlin, temos uma *feature* muito bacana chamada de **with()**!

O `with()` nos permite, a partir de um objeto, executar o código considerando tudo dentro de um escopo. Em outras palavras, com o objeto que queremos que é o `view.resumo_card_receita`, podemos executar tudo o que é relacionado a esse objeto dentro do escopo. Podemos pegar o `setTextColor(corReceita)` e executar diretamente dentro do escopo do `with()`, que será de fato, do `resumo_card_receita`.

```
    fun adicionaReceita() {
        var totalReceita = resumo().receita()

        with(view.resumo_card_receita){
            setTextColor(corReceita)
        }

        view.resumo_card_receita
        view.resumo_card_receita.text = totalReceita.formataParaBrasileiro()
    }
```

Como também podemos executar a sua *property*. Assim, não precisamos chamar o objeto várias vezes, apenas uma única vez dentro do `with()`.

```
    fun adicionaReceita() {
        var totalReceita = resumo().receita()

        with(view.resumo_card_receita){
            setTextColor(corReceita)
            text = totalReceita.formataParaBrasileiro()
        }
    }
```

Vamos testar?

A *app* não sofreu alterações. Então, dessa maneira, chamamos apenas uma única vez o objeto e conseguimos executar todas as operações com esse objeto, uma única vez.

Faremos essa modificação nos outros componentes.

```
fun adicionaDespesa() {
    var totalDespesa = resumo().despesa()
    with(view.resumo_card_despesa){
        setTextColor(corDespesa)
        text = totalDespesa.formataParaBrasileiro()
    }
}
```

Agora, nós podemos estar fazendo esse processo em `adicionaTotal()`. Repare que dentro dele, temos o mesmo processo, que é chamar o mesmo código dentro do `if()`, sendo que na verdade, da para nós colocarmos a chamada do código fora dele. Por exemplo:

```
fun adicionaTotal() {
    var total = resumo().total()
    var cor = if(total.compareTo(BigDecimal.ZERO) >= 0){
        corReceita
    } else {
        corDespesa
    }
    view.resumo_card_total.setTextColor(cor)
    view.resumo_card_total.text = total.formataParaBrasileiro()
}
```

Aqui, nós pedimos para a *property* setar uma cor, baseando-se na variável `cor` que irá conter o resultado do `if()`. Quando temos o **if expression**, conseguimos simplificá-lo, e fazer com que ele se torne uma *chamada de função*.

Vamos começar o processo de refatoração, e deixar toda a lógica dentro de uma função. Selecionamos o `if()` e todo o seu escopo, e utilizamos o atalho "Ctrl + Alt + M" pra *extrair uma função*. O nome dessa função será `corPor()`. A `corPor()` será baseada-se em um valor. Mandaremos a variável `total`, entretanto mudaremos o seu nome de `total` para `valor` utilizando o atalho "Shift + F6":

```
fun adicionaTotal() {
    var total = resumo().total()
    var cor = corPor(total)
    view.resumo_card_total.setTextColor(cor)
    view.resumo_card_total.text = total.formataParaBrasileiro()
}

private fun corPor(valor: BigDecimal): Int {
    return if (valor.compareTo(BigDecimal.ZERO) >= 0) {
        corReceita
    } else {
        corDespesa
    }
}
```

Nosso código está bem mais resumido, e temos como melhora-lo ainda mais.

Estávamos desconsiderando a questão de utilizar um *if expression*, sendo que podemos usar um **return**.

```
fun adicionaTotal() {
    var total = resumo().total()
    var cor = corPor(total)
    view.resumo_card_total.setTextColor(cor)
    view.resumo_card_total.text = total.formataParaBrasileiro()
}

private fun corPor(valor: BigDecimal): Int {
    if (valor.compareTo(BigDecimal.ZERO) >= 0) {
        return corReceita
    }
    return corDespesa
}
```

Ficou bem mais simples, não é mesmo? Executaremos com "Alt + Shift + F10". Legal! O *app* ainda está mantendo as informações. Então, conseguimos refatorar essa parte do nosso código. No entanto, ainda podemos fazer melhorias.

Dentro da função `adicionaTotal()`, ainda temos duas chamadas de objeto. Então, podemos fazer um `with()` novamente.

```
fun adicionaTotal() {
    var total = resumo().total()
    var cor = corPor(total)
    with(view.resumo_card_total) {
        setTextColor(cor)
        text = total.formataParaBrasileiro()
    }
}
```

Executaremos novamente para testar. É muito importante quando estamos em processo de refatoração, que executemos o código a cada mudança, pois isso evitará erros que poderão surgir mais tarde.

Para finalizar essa parte da refatoração, repare que no `compareTo()` da função `corPor()`, o Android Studio está nos dando uma sugestão. Com o atalho "Alt + Enter" conseguimos visualizar essa sugestão, que é *substituir "compareTo()" pelo operador ">=" diretamente*. Se dermos um "Enter" nessa sugestão, teremos esse resultado:

```
private fun corPor(valor: BigDecimal): Int {
    if (valor >= (BigDecimal.ZERO)) {
        return corReceita
    }
    return corDespesa
}
```

Isso significa que podemos comparar os objetos diretamente pelos operadores lógicos. Isso acontece porque no Kotlin não há **tipos primitivos**. Tudo o que utilizamos aqui são **objetos**.

Por debaixo dos panos, a comparação `<=` é o próprio `compareTo()`. Assim como também a comparação `==`, significa o `equals()`. Então, na verdade, tudo o que usamos dentro do Kotlin são *objetos*.

Então, conseguimos fazer a refatoração do código, e o `ResumoView()` ficou bem mais sucinto.