09

# Aplicando o total no resumo

### Transcrição

Conseguimos colocar as somas tanto de receita quanto de despesa. Nosso próximo passo é começarmos o processo de introduzir o **total** no Resumo.

Ciraremos uma função responsável por *calcular o Total* em `ResumoView()`:

```
class ResumoView(private val view: View,
                 private val transacoes: List<Transacao>) {

    fun adicionaReceita() {
        // lógica do for each
    }

    fun adicionaDespesa() {
        // lógica do for each
    }

    fun adicionaTotal() {

    }
}
```

Para fazer com que a função adicione o total no resumo, precisaremos do `totalReceita` e do `totalDespesa`, que estão localizados dentro das funções `adicionaReceita()` e `adicionaDespesa()`. Uma das alternativas é selecionar o código que faz o cálculo da despesa e da receita e *isolar-lo* em uma função menor.

A ideia é criar funções menores que já calculassem os valores de `totalDespesa` e de `totalReceita`. E a partir dessas funções, pegaríamos os valores. Entretanto, repare que estamos dentro de uma classe chamada `ResumoView`, e a princípio a responsabilidade dessa classe é de **apenas** mandar as informações pra View. Ou seja, ela não deve ter a responsabilidade de realizar cálculos.

Aplicaremos o processo de **delegar a responsabilidade** para uma classe específica.

Criaremos a nova classe no pacote "app > java > br.com.alura.financask > model" chamada de `Resumo`, pois será uma classe responsável por fazer todo o resumo, incluindo o cálculo de despesas, receitas e o total.

Para criar essa classe, selecionamos o pacote "model" e utilizamos o atalho "Alt + Insert". Chamaremos essa classe de *Resumo*.

Dentro da nova classe `Resumo`, implementaremos o cálculo da **receita** com a função `receita()`.

```
class Resumo {
    fun receita() {


    }
}
```

Pegaremos todo o código que faz o cálculo e colocaremos dentro do método `receita()`. Observe que dentro do for, utilizamos uma lista de transações que não está sendo compilada. Para resolver isso, é necessário passar essas transações como parâmetro do método e dizer que elas fazem parte de uma lista de transação.

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

Assim, basta devolvermos o valor da variável `totalReceita`, não nos esquecendo de que a função retorna um `BigDecimal`

```
class Resumo {
    fun receita(transacoes: List<Transacao>) : BigDecimal {
        var totalReceita = BigDecimal.ZERO
        for (transacao in transacoes) {
            if (transacao.tipo == Tipo.RECEITA) {
                totalReceita = totalReceita.plus(transacao.valor)
            }
        }
        return totalReceita
    }
}
```

Para utilizar essa função, podemos apagar o código copiado de `ResumoView`, instanciar um objeto do tipo `Resumo` e chamar a função que retorna o cálculo, no caso a função `receita()`.

```
    fun adicionaReceita() {
        var totalReceita = Resumo().receita(transacoes)
        view.resumo_card_receita.text = totalReceita.formataParaBrasileiro()
    }
```

Como nós não modificamos o valor da variável `totalReceita`, podemos voltar com o tipo `val`.

```
    fun adicionaReceita() {
        val totalReceita = Resumo().receita(transacoes)
        view.resumo_card_receita.text = totalReceita.formataParaBrasileiro()
    }
```

Para termos a certeza de que tudo está funcionando, executaremos a *app* com "Alt + Shift + F10" e ver se está tudo certo.

Como podemos ver, o código ainda está calculando, e de uma forma bem mais resumida, ou seja, não estamos realizando o cálculo dentro de `ResumoView`, pois isso não é de sua responsabilidade. Conseguimos isolar e delegar a

responsabilidade para uma nova classe chamada `Resumo`.

Assim como fizemos para a **receita**, faremos também para a **despesa**.

Copiaremos o código que calcula o total da despesa, e dentro de `Resumo`, criaremos a função `despesa()`, e em seguida colaremos o código copiado.

```
class Resumo {

    fun receita(transacoes: List<Transacao>) : BigDecimal {
            // for
        return totalReceita
    }

    fun despesa(transacoes: List<Transacao>) : BigDecimal {
        var totalDespesa = BigDecimal.ZERO
        for (transacao in transacoes){
            if (transacao.tipo == Tipo.DESPESA) {
                totalDespesa = totalDespesa.plus(transacao.valor)
            }
        }
        return totalDespesa
    }
}
```

Perceba que sempre que chamamos tanto a função `receita()` quanto a função `despesa()`, somos obrigados a passar a lista de transações como parâmetro. Existem alguns detalhe engraçado sobre passar essa lista como parâmetro. Por exemplo, se passarmos uma lista de cinco transações na `receita()`, será feito o cálculo em cima das cinco transações. Se passarmos uma lista de dez transações na `despesa()`, será feito o cálculo em cima das dez transações. Ou seja, corremos o risco de passarmos **listas diferentes**.

Faz todo o sentido mandar essa lista de transações via **construtor primário** e criar uma *property* baseando-se nela, e então, os métodos usarem essa *property* como referência para fazer esse cálculo. Dessa maneira, garantimos que a *mesma* lista de transações será usada, tanto para calcular a receita, quanto para calcular a despesa.

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

    fun receita() : BigDecimal {
        // for
        return totalReceita
    }

    fun despesa() : BigDecimal {
        // for
        return totalDespesa
    }
}
```

Deixamos como `private`, pois não faz sentido deixarmos como `public` e aí todos tiverem acesso a ela.

Mudaremos em `ResumoView`, a parte do cálculo de despesa. Agora usaremos o método de `despesa()`:

```
class ResumoView(private val view: View,
                 private val transacoes: List<Transacao>) {

    fun adicionaReceita() {
        var totalReceita = Resumo(transacoes).receita()
        view.resumo_card_receita.text = totalReceita.formataParaBrasileiro()
    }

        fun adicionaDespesa() {
        var totalDespesa = Resumo(transacoes).despesa()
        view.resumo_card_despesa.text = totalDespesa.formataParaBrasileiro()
    }

    fun adicionaTotal() {

    }
}
```

Se observarmos melhor, podemos reparar que estamos enviando instâncias diferentes! E com isso, estamos correndo o risco de mandar listas diferentes, e se acontecer isso, os cálculos serão diferentes!

Por isso, faz todo o sentido criar uma *property* que irá manter essa instância que será a referência para calcular tanto a receita quanto a despesa. Antes das funções, criaremos essa *property* do tipo `Resumo`. A inicialização dessa *property* será uma instância de `Resumo()` que receberá as `transacoes`.

```
class ResumoView(private val view: View,
                 transacoes: List<Transacao>) {

    private val resumo: Resumo = Resumo(transacoes)

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

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

    fun adicionaTotal() {

    }
}
```

Os métodos passaram a pegar o objeto `resumo` que é a *property*. Assim, evitamos vários problemas com lista errada de transações. Ao fazer isso, passamos a não usar mais a referência do `BigDecimal`. Com o atalho "Ctrl + Alt + O", ajustamos os imports, ou seja, se estiver faltando algum, este será acrescentado, ou se estiver sobrando algum que não está sendo usado, o mesmo será apagado.

Conseguimos calcular a receita e a despesa, agora falta calcularmos o **total** dentro da classe que realiza os cálculos, a `Resumo`.

A função `total()` é a subtração da receita pela despesa. Para isso, utilizamos a função `subtract()` para subtrair.

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

    fun receita() : BigDecimal {
        // for
        return totalReceita
    }

    fun despesa() : BigDecimal {
        // for
        return totalDespesa
    }

    fun total() : BigDecimal {
        return receita().subtract(despesa())
    }
}
```

Desse modo, a função que calcula o total está pronta! Já na classe `ResumoView`, diremos que a função `adicionaTotal()` tem uma variável `total` que recebe o valor retornado da função `total()`.

```
class ResumoView(private val view: View,
                 transacoes: List<Transacao>) {

    private val resumo: Resumo = Resumo(transacoes)

    fun adicionaReceita() {
        val totalReceita = resumo.receita()
        view.resumo_card_receita.text = totalReceita.formataParaBrasileiro()
    }

    fun adicionaDespesa() {
        val totalDespesa = resumo.despesa()
        view.resumo_card_despesa.text = totalDespesa.formataParaBrasileiro()
    }

    fun adicionaTotal() {
        val total = resumo.total()
        view.resumo_card_total.text =  total.formataParaBrasileiro()
    }
}
```

Agora que ja temos o `adicionaTotal()`, vamo chamá-lo na *activity* `ListaTransacoesAcivity` dentro da função `configuraResumo()`:

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

Vamos executar!

Legal! Agora nós somos capazes de colocar a receita, colocar a despesa, e realizar o cálculo e apresentá-lo no painel.