06

Criando a classe para o componente resumo

Transcrição

Até agora, conseguimos configurar o valor da *Receita* no Resumo das transações. Mas perceba que aqui, temos uma responsabilidade a mais na *activity*, onde antes era somente configurar a lista de transações. Agora, ela também é responsável por configurar o Resumo.

Em outras palavras, faz todo o sentido realizar uma refatoração na qual irá delegar toda essa responsabilidade para uma classe específica, a fim de computar as informações do Resumo.

Como podemos mandar uma responsabilidade para outras classe?]

Dentro do projeto, acessaremos "app > java > br.com.alura.financask > ui", e criar uma classe que será responsável por cuidar da View do Resumo. Portanto, usaremos o "Alt + Insert" para criar um arquivo novo, selecionaremos "Kotlin File/Class", e a chamaremos de `ResumoView`.

Relembrando, essa classe ficará responsável por colocar todas as informações que teremos da View, em relação ao Resumo.

```
package br.com.alura.financask.ui

class ResumoView {

}
```

A ideia é copiar o comportamento do `adicionaReceitaNoResumo()`, e colar dentro da classe `ResumoView`. Realizamos os imports necessários, e então conseguimos fazer com que o `adicionaReceitaNoResumo()` fique em uma classe responsável. Portanto, não precisamos mais dele no momento. Podemos apagá-lo de nossa *activity*.

Em seguida, dentro de `onCreate()`, faremos uma instância de `ResumoView`. A partir dessa instância, chamaremos o `adicionaReceitaNoResumo()`.

```
class ListaTransacoesActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_lista_transacoes)

        val transacoes: List<Transacao> = transacoesDeExemplo()

        ResumoView().adicionaReceitaNoResumo(transacoes)

        configuraLista(transacoes)
    }
}
```

Não podemos nos esquecer que o método `adicionaReceitaNoResumo()` está como **private**. Removemos o modificador desse método na classe `ResumoView`, e assim a função torna-se pública.

Ainda dentro da classe `ResumoView`, vamos resolver um problema de compilação. Veja que neste processo, dentro da função `adicionaReceitaNoResumo()` precisamos acessar o componente `resumo_card_receita`, ou seja, precisamos fazer uso do Synthetic. Mas, isso não é possível, pois `ResumoView` não é uma *activity* que tem o poder de pegar todas as propriedades de um layout e transformar em uma propriedade dela.

Em outras palavras, precisamos de uma **referência** de uma `View`, assim como fizemos no *adapter*.

Então, podemos falar que o `ResumoView()` irá receber, via construtor, uma *property* privada que será uma *view* do tipo `View`.

```
class ResumoView(private val view: View) {

}
```

E agora que temos essa view, podemos chamar o componente que precisamos, apenas importando:

```
class ResumoView(private val view: View) {

    fun adicionaReceitaNoResumo(transacoes: List<Transacao>) {
        var totalReceita = BigDecimal.ZERO
        for (transacao in transacoes){
            if (transacao.tipo == Tipo.RECEITA) {
                totalReceita = totalReceita.plus(transacao.valor)
            }
        }
        view.resumo_card_receita.text = totalReceita.formataParaBrasileiro()
    }
}
```

Com o atalho "Ctrl + Alt + O", conseguimos remover os imports que não estão mais sendo usados no projeto.

Recapitulando o que fizemos: para poder pegar o componente `resumo_card_receita`, declaramos um objeto do tipo `View` para ter acesso a ele.

Agora, é a vez de ver o que está acontecendo de errado com a *activity* `ListaTransacoesActivity`.

Perceba que temos um problema de compilação, pois o `ResumoView()` **exige** um objeto do tipo `View`.

Como podemos pegar um objeto do tipo View aqui na *activity*?

A activity possui uma *property* que nos permite pegar uma janela na qual ela representa a aplicação para nós. Bom, então logicamente usaríamos a *property* chamada **window**, certo?... Errado!

Esse `window` que aparece para nós na IDE, é um objeto do tipo `Window`, e o que precisamos é de uma View! Para pegar uma *View* do *window*, podemos pegar a partir de uma "view decorada":

```
class ListaTransacoesActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_lista_transacoes)

        val transacoes: List<Transacao> = transacoesDeExemplo()
        window.decorView
        ResumoView().adicionaReceitaNoResumo(transacoes)

        configuraLista(transacoes)
    }
}
```

A funcionalidade representa de fato, a tela da *activity*, sendo assim uma das formas de se pegar uma **view** da *activity*. Inclusive, podemos extraí-la para uma variável, acrescentando `.val` :

```
window.decorView.val
```

E depois, podemos nomeá-la de `view` :

```
class ListaTransacoesActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_lista_transacoes)

        val transacoes: List<Transacao> = transacoesDeExemplo()

        val view = window.decorView
        ResumoView(view).adicionaReceitaNoResumo(transacoes)

        configuraLista(transacoes)
    }
}
```

Então o `decorView` nos devolve uma `View` .

Após essa refatoração, vamos ver se ainda está tudo funcionando? Utilizaremos o comando "Alt + Shift + F10".

Legal, tudo está funcionando perfeitamente. Mas agora estamos delegando toda a responsabilidade para o `ResumoView` .

Podemos fazer um outro processo de refatoração aqui.

Nas linhas abaixo, estamos configurando o Resumo:

```
val view: View = window.decorView
ResumoView(view).adicionaReceitaNoResumo(transacoes)
```

Selecionamos essas linhas, e utilizamos o atalho "Ctrl + Alt + N" para *extrair uma função*, onde que, é dentro dessa função que colocaremos todo o comportamento para configurar a View de `Resumo`. Colocaremos o nome de `configuraResumo`.

```
class ListaTransacoesActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_lista_transacoes)

        val transacoes: List<Transacao> = transacoesDeExemplo()

        configuraResumo(transacoes)

        configuraLista(transacoes)
    }
}
```

Agora que estamos conseguindo adicionar a receita, podemos começar com o processo de adicionar a **despesa**. Dentro da `ResumoView`, criaremos a função `adicionaDespesaNoResumo()` que ficará responsável por adicionar uma nova *Despesa*.

```
class ResumoView(private val view: View) {

    fun adicionaReceitaNoResumo(transacoes: List<Transacao>) {
        var totalReceita = BigDecimal.ZERO
        for (transacao in transacoes){
            if (transacao.tipo == Tipo.RECEITA) {
                totalReceita = totalReceita.plus(transacao.valor)
            }
        }
        view.resumo_card_receita.text = totalReceita.formataParaBrasileiro()
    }

    fun adicionaDespesaNoResumo(transacoes: List<Transacao>) {
        var totalDespesa = BigDecimal.ZERO
        for (transacao in transacoes){
            if (transacao.tipo == Tipo.DESPESA) {
                totalDespesa = totalDespesa.plus(transacao.valor)
            }
        }
        view.resumo_card_despesa.text = totalDespesa.formataParaBrasileiro()
    }
}
```

Legal, a função para a despesa está criada. Mas, você não concorda que o nome da função não precisa ser `adicionaDespesaNoResumo()`? Aliás, não é necessário nos métodos referentes a Receita e a Despesa, pois ambas as funções estão **adicionando** coisas dentro do *Resumo*, então podemos remover a parte `NoResumo` do nome desses métodos.

Não teremos problemas em fazer essa mudança no método `adicionaDespesa()` pois ele acabou de ser criado. Entretanto, não podemos simplesmente alterar o nome do primeiro método para `adicionaReceita()`, pois a *activity* utiliza essa função, e então ela irá parar de compilar.

Podemos utilizar um recurso para modificar uma função que já está sendo usada. Com o cursor em cima da função, utilize o comando "Shift + F6", e então podemos remover a parte `...NoResumo`. Dessa forma, mantemos a compilação e batemos em todos os pontos que estavam utilizando essa função.

Vamos começar com a lógica!

Como fizemos, ao invés de colocar no componente `resumo_card_receita`, colocamos no componente `resumo_card_despesa`.

Repare que ainda não estamos utilizando a função `adicionaDespesa()` para realmente adicionar uma despesa. Se executarmos o projeto agora, não aparecerá nenhuma despesa no Resumo. Em outras palavras, é necessário chamar essa função na *activity* e chamá-la.

Pegaremos o objeto do `ResumoView(view)` para reutilizá-lo.

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

Vamos ver se está tudo funcionando corretamente, executando o comando "Alt + Shift + F10".

Olhe só:

Como podemos ver, agora temos um valor referente as Despesas, onde foi somado uma despesa indefinida no valor de `R$ 200,00` mais uma outra despesa de almoço no valor de `R$ 20,50` .

Vamos voltar ao código. Todas as vezes que chamamos a função `adicionaReceita()` ou `adicionaDespesa()` , estamos mandando a lista de transações. De fato, os dois métodos estão utilizando a mesma lista. Então, faz todo o sentido enviarmos essa lista no **construtor primário**. Dessa forma, ele será transformado em uma *property* que será acessível para todos os métodos, eliminando ter que enviar essa lista como um parâmetro todas as vezes que queremos adicionar uma Receita ou Despesa.

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

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

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

Assim feito, podemos apagar o parâmetro `transacoes: List<Transacao>` dos métodos da Receita e da Despesa. A partir de agora, as transações serão baseadas na nova *property*.

Ao voltarmos para a *activity*, vemos alguns problemas de compilação. Isso ocorre, pois é preciso passar `transacoes` como parâmetro de `ResumoView` .

```
class ListaTransacoesActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_lista_transacoes)

        val transacoes: List<Transacao> = transacoesDeExemplo()

        configuraResumo(transacoes)

        configuraLista(transacoes)
    }

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

Testaremos o código agora com "Alt + Shift + F10".

As informações foram mantidas, e isso nos diz que o código está funcionando corretamente. Com isso, conseguimos adicionar tanto a Receita quanto a Despesa de uma maneira bem mais elegante.