01

Enviando o Dialog para uma classe específica

Transcrição

Atualmente, o usuário consegue adicionar receitas na lista de transações. Mas, e as despesas? Será que ele irá conseguir adicioná-las ao clicar no botão "Adicionar despesa"? No momento, nada acontece. A nossa *app* também precisa permitir que o usuário adicione despesas, e é justamente isso que faremos agora.

Ao observar o código para adicionar o *Dialog* para a adição de uma nova receita, percebemos que ele ficou um pouco grande.

Na *app* base, ao adicionar uma nova despesa, temos praticamente o mesmo comportamento que temos na receita. A única coisa diferente entre elas, são os parâmetros que indicam que agora temos uma *despesa*. Então, antes mesmo de implementarmos essa funcionalidade, faremos uma refatoração para permitir que reutilizemos o mesmo código da receita para a despesa.

Qual é o primeiro passo?

Uma boa parte do código da `ListaTransacoesActivity` é somente para configurar o *Dialog*, sendo assim vamos extraí-lo para uma função desde `val view: View = window.decorView` até a linha `.show()`, e essa nova função será chamada de `configuraDialog` utilizando o atalho "Ctrl + Alt + M".

```
class ListaTransacoesActivity : AppCompatActivity() {

    private val transacoes: List<Transacao> = listOf()

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

        configuraResumo()

        configuraLista()

        lista_transacoes_adiciona_receita
                .setOnClickListener {
                    configuraDialog()
                }
    }

    private fun configuraDialog() {
        // todo o código para a configuração do Dialog
    }
```

Como podemos ver, no componente `lista_transacoes_adiciona_receita`, tem o *Listener* que simplesmente chama a função `configuraDialog()` que faz a configuração do *Dialog*. Mas agora, entramos em um outro caso em que estamos fazendo com que a *Activity* tenha muita responsabilidade. Além de setar as informações da *view*, ela também é obrigada a configurar todos esses parâmetros do *Dialog*.

Isso significa que o código do *Dialog* precisa ser delegado para uma classe específica, que será responsável por montar o *Dialog* para nós, da mesma maneira que fizemos em *ResumoView*.

Vamos acessar o pacote raiz do nosso projeto. Depois, acessaremos "UI". Já que todo esse código está destinado a uma configuração de *Dialog*, então criaremos um pacote chamado "Dialog". Dentro desse pacote criaremos a classe `AdicionaTransacaoDialog`, onde será permitido criar um *Dialog* que adiciona transações.

Recortaremos o código extraído da `ListaTransacoesActivity` para a classe `AdicionaTransacaoDialog`. No momento em que colamos o código, temos que realizar vários **imports** que estamos utilizando la na *activity*. Não se preocupe, o código apresentará diversos erros de compilação, e iremos tratá-los agora.

O primeiro passo para ajustar essa refatoração é **separar cada bloco em funções menores**! Essa é uma das *técnicas primárias* para conseguirmos melhorar o nosso código. Dê uma olhada no primeiro bloco:

```
val view: View = window.decorView
val viewCriada = LayoutInflater.from(context: this)
        .inflate(R.layout.form_transacao,
                view as ViewGroup,
                attachToRoot: false)
```

Aqui, o objetivo é **criar o *layout***. Portanto, podemos criar uma função no final do código, a `criaLayout()` para armazenar todo o código mostrado acima.

```
class AdicionaTransacaoDialog {

    private fun configuraDialog() {
        // código que cria o Dialog
    }

    private fun criaLayout() {
        val view: View = window.decorView
        val viewCriada = LayoutInflater.from(context: this)
        .inflate(R.layout.form_transacao,
                view as ViewGroup,
                attachToRoot: false)
    }
}
```

Isolamos o código que cria o Layout! Agora, analisaremos esse código para ver o que podemos modificar de acordo com o que tínhamos na *activity* e o que não teremos mais aqui nessa classe. A primeira etapa que podemos analisar é o que de fato, conseguimos ter acesso dentro da classe.

Repare que o `window.decorView`, é a *property* que tem na *activity*, mas aqui não temos acesso à ela. Precisávamos dessa *property* para ter o objeto `view`, e conseguir mandar o `ViewGroup` para o `Inflate()`. Nós não precisamos mais desse objeto dentro dessa classe. Podemos fazer com que ela peça um **ViewGroup** direto do construtor dessa classe.

```
class AdicionaTransacaoDialog(private val viewGroup: ViewGroup) {

    private fun configuraDialog() {
        // código que cria o Dialog
    }
```

```
    private fun criaLayout() {

        val viewCriada = LayoutInflater.from(context: this)
            .inflate(R.layout.form_transacao,
                viewGroup,
                attachToRoot: false)
    }
}
```

Assim nós já temos acesso ao `ViewGroup`. E ele será mandado através do construtor por padrão, com isso não será mais necessário fazer a conversão.

Além da parte do `viewGroup`, temos o `this` que se refere ao *context* da *activity*. Agora, ao invés de pedir o `this`, pediremos o **context**, que virá do construtor.

```
class AdicionaTransacaoDialog(private val viewGroup: ViewGroup,
                              private val context: Context) {

    private fun configuraDialog() {
        // código que cria o Dialog
    }

    private fun criaLayout() {
        val viewCriada = LayoutInflater.from(context)
            .inflate(R.layout.form_transacao,
                viewGroup,
                attachToRoot: false)
    }
}
```

O nosso código está mais expressivo no que está acontecendo e refatorado. Para melhorar ainda mais, ao invés de devolver `viewCriada`, devolveremos o `LayoutInflater`.

```
    private fun criaLayout() : View {
        return LayoutInflater.from(context)
            .inflate(R.layout.form_transacao,
                viewGroup,
                attachToRoot: false)
    }
```

O que irá mudar no `configuraDialog()`? Bom, não iremos mais precisar da linha que contém o `window.decorView`, e também não precisamos mais criar o `LayoutInflater` novamente. Basta apenas chamar o `criaLayout()`.

```
class AdicionaTransacaoDialog(private val viewGroup: ViewGroup,
                              private val context: Context) {

    private fun configuraDialog() {

        val viewCriada = criaLayout()

        val ano = 2017
```

```
        val mes = 9
        val dia = 18
        // código que cria o Dialog
    }

    private fun criaLayout() : View {
        return LayoutInflater.from(context)
            .inflate(R.layout.form_transacao,
                viewGroup,
                attachToRoot: false)
    }
}
```

Após ter feito o primeiro passo, já podemos analisar o próximo bloco de código. Analise desde a linha que contém `val ano = 2017` até `.show()` que se encontra abaixo das variáveis `ano, mes, dia` do `DatePickerDialog()`.

O objetivo dessa parte do código é **configurar o campo da data**. Portanto, vamos selecionar toda essa parte, e utilizar o atalho no Android Studio "Ctrl + Alt + M", assim extrairemos para uma função `configuraCampoData(viewCriada)`. Nesse momento, todo aquele código selecionado, foi mandado para a função `configuraCampoData()` no final do código.

```
class AdicionaTransacaoDialog(private val viewGroup: ViewGroup,
                              private val context: Context) {

    private fun configuraDialog() {

        val viewCriada = criaLayout()

        configuraCampoData(viewCriada)

        // restante do código que cria o Dialog
    }

    private fun criaLayout() : View {...}

    private fun configuraCampoData(viewCriada: View ) {
        val ano = 2017
        val mes = 9
        val dia = 18

        val hoje = Calendar.getInstance()
        viewCriada.form_transacao_data
                .setText(hoje.formataParaBrasileiro())
        viewCriada.form_transacao_data
                .setOnClickListener {
                    DatePickerDialog(context: this,
                            DatePickerDialog.OnDateSetListener {...}
                            , ano, mes, dia)
                            .show()
                }
    }

    private fun criaLayout() : View {...}
}
```

Vamos dar uma olhada no código que faz a criação do campo `data`, e ver o que pode ser melhorado. Perceba que estávamos colocando informações de `data` com **valores físicos**. Então, ao invés de colocar essa informação fixa, iremos pegar o **ano**, **mes**, e o dia a partir de uma propriedade da variável `hoje`:

```
private fun configuraCampoData(viewCriada: View ) {
        val hoje = Calendar.getInstance()

        val ano = hoje.get(Calendar.YEAR)
        val mes = hoje.get(Calendar.MONTH)
        val dia = hoje.get(Calendar.DAY_OF_MONTH)
    }
```

Pegando o valor das datas dessa forma, não importa o dia que seja, sempre irá mudar, sempre será o dia de hoje. Repare também que temos o `context: this` novamente a diante. Então, trocaremos para a *property* que criamos `context`. Dentro da *expressão Lambda*, veja que `view` não está sendo usada, e o Android Studio nos dá a sugestão de renomeá-la para **_**. E por que ele nos sugere isso?

Quando colocamos um parâmetro com esse nome, indicamos que não utilizaremos esse parâmetro dentro do escopo do Lambda neste momento. Caso venhamos a utilizá-lo mais tarde, é só renomeá-lo.

```
    viewCriada.form_transacao_data
            .setText(hoje.formataParaBrasileiro())
    viewCriada.form_transacao_data
            .setOnClickListener {
                DatePickerDialog(context,
                        DatePickerDialog.OnDateSetListener { _, ano, mes, dia ->
                            val dataSelecionada = Calendar.getInstance()
                            dataSelecionada.set(ano, mes, dia)
                            viewCriada.form_transacao_data
                                .setText(dataSelecionada.formataParaBrasileiro()
                        }
                        , ano, mes, dia)
                        .show()
            }
```

Vamos voltar para o `configuraDialog()`. O próximo bloco de código a ser refatorado é o que configura o campo de categoria, e ele começa da linha `val adapter = ArrayAdapter` e termina em `viewCriada.form_transacao_categoria.adapter = adapter`. Ao selecionar todo esse bloco, usaremos o atalho "Ctrl + Alt + M" e vamos extrair para a função `configuraCampoCategoria()`. Não podemos nos esquecer de modificar a referência do `context`:

```
class AdicionaTransacaoDialog(private val viewGroup: ViewGroup,
                              private val context: Context) {

    private fun configuraDialog() {

        val viewCriada = criaLayout()

        configuraCampoData(viewCriada)

        configuraCampoCategoria(viewCriada)
```

```
        // código do AlertDialog.Builder
    }

    private fun configuraCampoCategoria(viewCriada: View) {
        val adapter = ArrayAdapter
            .createFromResource(context,
                R.array.categorias_de_receita,
                android.R.layout.simple_spinner_dropdown_item)

        viewCriada.form_transacao_categoria.adapter = adapter

    }
```

Repare um detalhe interessante. A variável `viewCriada` está sendo utilizada em diversos pontos do código. Portanto, se diversos membros estão utilizando a mesma que foi criada dentro do `configuraDialog()`, faz todo o sentido transformá-la em uma **property**.

Então, é necessário tirar essa variável do escopo `configuraDialog()`. Para mover uma linha para cima ou para baixo dentro do Android Studio, usamos "Ctrl + Shift + seta (para cima ou para baixo)". Mas, esse atalho não deixa que uma linha saia do escopo de uma função. Sendo assim só conseguiremos mover a linha que contém a `viewCriada` caso utilizemos o comando "Alt + Shift + seta (para cima ou para baixo)".

Para ninguém mais ter acesso a ela, colocaremos o modificador `private`.

```
class AdicionaTransacaoDialog(private val viewGroup: ViewGroup,
                              private val context: Context) {

    private val viewCriada = criaLayout()

    private fun configuraDialog() {...}
```

Agora, não é mais necessário mandar `viewCriada` via parâmetro. Podemos retirá-la de alguns pontos do código, inclusive das funções que criamos no final do código.

```
    private fun configuraDialog() {
        configuraCampoData()
        configuraCampoCategoria()
        // restante do código que cria o Dialog
    }

    private fun configuraCampoCategoria() {...}

    private fun configuraCampoData() {...}
```

Vamos ver agora, o próximo bloco a ser refatorado: o `AlertDialog`. O *AlertDialog* realiza uma configuração para que nós venhamos ter o formulário. Podemos dizer que, o bloco de código que começa na linha `AlertDialog.Builder(context: this)` e termina em `.show()` logo abaixo de `.setNegativeButton()`, vai configurar o formulário! Utilizaremos o atalho "Ctrl + Alt + M" e chamaremos essa nova função de `configuraFormulario()`.

Desta maneira, conseguimos fazer a extração das "três extremidades" que haviam em nosso código, o `configuraCampoData()`, `configuraCampoCategoria()` e `configuraFormulario()`.

```
private fun configuraDialog() {
    configuraCampoData()
    configuraCampoCategoria()
    configuraFormulario()
}

private fun configuraFormulario() {...}

private fun configuraCampoCategoria() {...}

private fun configuraCampoData() {...}
```

O que falta é fazer algumas modificações em `configuraFormulario()`. O primeiro passo é retirar o `this`, e deixar somente o `context`.

```
private fun configuraFormulario() {
    AlertDialog.Builder(context)
        .setTitle()
        .setView()
        .setPositiveButton()
}
```

Em seguida, podemos observar alguns parâmetros que não utilizamos, e sendo assim, acrescentaremos o *underscore* `_`:

```
.setPositiveButton(text: "Adicionar",
    { _, _ ->
        // variáveis em texto
    })
```

Modificaremos o `context` no **Toast**:

```
val valor = try {
    BigDecimal(valorEmTexto)
} catch (exception: NumberFormatExeption) {
    Toast.makeText(context,
            text: "Falha na conversão de valor",
            Toast.LENGTH_LONG)
            .show()
}
```

Certo, chegou a hora de fazer a refatoração dentro do `.setPositiveButton()`, e ver o quanto de código podemos diminuir.

Dentro de seu *Listener*, temos a parte que pega os valores em texto dos campos utilizados: o `form_transacao_valor`, o `form_transacao_data` e o `form_transacao_categoria`. Então, já que vamos **reutilizar** essas variáveis, não faz sentido realizar uma extração de código que se refere à elas, então podemos deixar como está atualmente. No bloco a seguir, contém o `try catch`. E nesse bloco de código, é preciso interpretar o que está acontecendo, sendo que na verdade, esse código tenta fazer uma **conversão do BigDecimal**.

Vamos copiar esse código, e então criar uma função chamada de `converteCampoValor()` e colar o código no escopo dessa nova função.

```
private fun converteCampoValor() {
    try{
        BigDecimal(valorEmTexto)
    } catch(exception: NumberFormatException) {
        Toast.makeText(context: this,
                text: "Falha na conversão de valor",
                Toast.LENGTH_LONG)
                .show()
        BigDecimal.ZERO
    }
}
```

Como parâmetro, esperamos um `valorEmTexto` . E no momento que realizarmos a conversão, retornaremos o `try catch` *expression*, indicando também que esperamos um *BigDecimal* de retorno.

```
private fun converteCampoValor(valorEmTexto: String) : BigDecimal {
    return try{
        BigDecimal(valorEmTexto)
    } catch(exception: NumberFormatException) {
        Toast.makeText(context,
                text: "Falha na conversão de valor",
                Toast.LENGTH_LONG)
                .show()
        BigDecimal.ZERO
    }
}
```

Ao observar o comportamento dessa função de pegar uma String e converter em BigDecimal, vemos que é bem comum em nossa *app*. Por isso, podemos até pensar em transformar todo esse código em *extension function*, mas repare que o `catch` é de grande importância nesse momento do código pois existe um `Toast` nele.

Então, deixaremos esse código dentro da classe `AdicionaTransacaoDialog` . Agora, o que precisamos fazer é chamar essa função na variável `valor` :

```
.setPositiveButton(text: "Adicionar",
    { _, _ ->
        val valorEmTexto = viewCriada
            .form_transacao_valor.text.toString()
        val dataEmTexto = viewCriada
            .form_transacao_valor.text.toString()
        val categoriaEmTexto = viewCriada
            .form_transacao_valor.selectedItem.toString()


        val valor = converteCampoValor(valorEmTexto)
    })
```

Em seguida, temos um bloco de **conversão de data**:

```
    val formatoBrasileiro = SimpleDateFormat(pattern: "dd/MM/yyyy")
    val dataConvertida: Date = formatoBrasileiro.parse(dataEmTexto)
    val data = Calendar.getInstance()
    data.time = dataConvertida
```

Entretanto, diferente daquele caso de conversão que utiliza o `Toast`, podemos criar uma *extension function* para esse bloco, uma função que converterá uma `String` para um `Calendar`. Na classe `AdicionaTransacaoDialog`, vamos criar a função:

```
fun String.converteParaCalendar() {

}
```

Então, assim que chamarmos a `converteParaCalendar()`, pegaremos a `dataEmTexto`, e faremos a conversão em *Calendar*. Copiaremos o código de conversão de data para dentro dessa função. Dentro da função, basta mudarmos a referência de `formatoBrasileiro.parse(dataEmTexto)` para `formatoBrasileiro.parse(this)`. Depois é só retornar a data

```
fun String.converteParaCalendar() : Calendar {
    val formatoBrasileiro = SimpleDateFormat(pattern: "dd/MM/yyyy")
    val dataConvertida: Date = formatoBrasileiro.parse(source: this)
    val data = Calendar.getInstance()
    data.time = dataConvertida
    return data
}
```

E assim criamos a nossa extensão da String. Ao invés de chamarmos todo esse código, o que podemos fazer para obter o valor da `data`? Então, para a **data**, pegaremos o `valorEmTexto` e converteremos para `Calendar`:

```
    .setPositiveButton(text: "Adicionar",
        { _, _ ->
            val valorEmTexto = viewCriada
                .form_transacao_valor.text.toString()
            val dataEmTexto = viewCriada
                .form_transacao_valor.text.toString()
            val categoriaEmTexto = viewCriada
                .form_transacao_valor.selectedItem.toString()

            val valor = converteCampoValor(valorEmTexto)

            val data =  dataEmTexto.converteParaCalendar()
        })
```

Novamente, é necessário extrair a função `converteParaCalendar()` para um lugar específico de *extensões*. Recortaremos toda a função, e colocaremos na extensão **StringExtension.kt**, localizado no pacote "extension".

```
fun String.limitaEmAte(caracteres: Int) : String {
    if(this.length > caracteres) {
        val primeiroCaractere = 0
        return "${this.substring(primeiroCaractere, caracteres)}..."
    }
```

```
    return this
}

fun String.converteParaCalendar() : Calendar {
    val formatoBrasileiro = SimpleDateFormat(pattern: "dd/MM/yyyy")
    val dataConvertida: Date = formatoBrasileiro.parse(source: this)
    val data = Calendar.getInstance()
    data.time = dataConvertida
    return data
}
```

Basta voltarmos ao *Dialog*, e importar. Como podemos ver, conseguimos resolver a parte das conversões em funções menores. O código a seguir cria uma transação:

```
val transacaCriada = Transacao(tipo = tipo.RECEITA,
    valor = valor,
    data = data,
    categoria = categoriaEmTexto)
```

Não é necessário extrair para uma função menor, pois já estamos utilizando o construtor da `Transacao`. Agora, entramos em um outro problema:

```
atualizaTransacoes(transacaoCriada)
lista_transacoes_adiciona_menu.close(true)
```

Esse código está diretamente ligado na *activity*. Então, como vamos fazer para executar todo esse código, que na verdade precisa ser executado na *activity*, dentro dessa classe?

Para resolver esse detalhe, a seguir veremos quais são as alternativas e o que cada uma delas acaba impactando para nós. Então, até mais!