01

Utilizando extension function

Transcrição

Na aula anterior colocamos as informações relacionadas às transações na aplicação, como categorias, valores e datas; para estas, implementamos um formatador brasileiro.

No entanto, reparamos que estamos colocando muita responsabilidade para o `getView`, pois além de lidar com as informações das *views*, ele está tendo que formatar datas, com que o *adapter* está se responsabilizando também.

Uma maneira ideal de resolver este tipo de situação é **extrair o comportamento para uma classe específica**, como o `Utils` por exemplo, mantendo-se o código mais coerente.

Isto é bastante comum quando fazemos código em Java. O formato brasileiro de data é bem importante para a nossa aplicação, e seria muito interessante se a própria classe `Calendar` conseguisse formatar isto.

No Kotlin, existe algo chamado *Extension function*, que permite que coloquemos comportamentos a mais em classes que não são nossas e, para fazermos isto, extrairemos todo o código referente à data para uma função. No fim de `ListaTransacoesAdapter`, teremos:

```
fun formataParaBrasileiro() {
    val formatoBrasileiro = "dd/MM/yyyy"
    val format = SimpleDateFormat(formatoBrasileiro)
    val dataFormatada = format.format(transacao.data.time)
}
```

Agora precisaremos informar a função de que ela não será chamada pelo nosso *adapter*, e sim pelo `Calendar`. Então, o código ficará desta forma:

```
fun Calendar.formataParaBrasileiro() {
    val formatoBrasileiro = "dd/MM/yyyy"
    val format = SimpleDateFormat(formatoBrasileiro)
    val dataFormatada = format.format(transacao.data.time)
}
```

Faremos o `import` indicando o calendário que queremos - no caso, o do `java.util`, que é o que estamos utilizando. Com uma *Extension function*, temos acesso internamente a um objeto do tipo `Calendar`, que podemos utilizar por meio da *keyword* `this` quando chamamos a data da transação, em vez de chamar a data da transação em si:

```
val dataFormatada = format.format(this.time)
```

A partir disto, todo objeto `Calendar` que for chamar `formataParaBrasileiro()` irá se autoformatar, atingindo o resultado desejado. Falta devolvermos o retorno deste *format*, ou seja, a data formatada, a variável que criamos.

Como ela é uma *string*, indicaremos que estaremos devolvendo uma *string* também:

```
fun Calendar.formataParaBrasileiro() {
    val formatoBrasileiro = "dd/MM/yyyy"
    val format = SimpleDateFormat(formatoBrasileiro)
    val dataFormatada = format.format(this.time)
    return dataFormatada
}
```

Deste modo, temos uma função sendo estendida à função `Calendar`. Para fazer sua chamada, usávamos `dataFormatada`, agora, é possível deixarmos assim:

```
viewCriada.transacao_data.text = transacao.data.formataParaBrasileiro()
```

Vamos testar a app com "Alt + Shift + F10"! O formato se mantém inalterado. Notem que `formataParaBrasileiro` não é mais uma função de uma classe qualquer, e sim de `Calendar`, do `java.util`. Inclusive, se tentarmos chamar `formataParaBrasileiro`, ele nem é reconhecido, tampouco aparece no modo de autocompletar, pois esta chamada só será feita com o objeto do tipo `Calendar`.

O `formatoBrasileiro` se encontra dentro do *adapter*, não muito apropriado para isto. Precisaremos, então, isolar esta função, pois ela será reutilizada em outros momentos, então não faz nenhum sentido ter que fazer *import* do *adapter* para poder utilizá-la.

No pacote raiz, "br.com.alura.financask", criaremos outro, denominado "extension", a partir do qual poderemos colocar todas as extensões de classes existentes que sentimos necessidade de incluir, como foi o caso do `formatoBrasileiro`.

Criaremos um novo arquivo em Kotlin, não necessariamente uma classe, denominada "CalendarExtension", para onde copiaremos e colaremos toda a função do `Calendar` presente no *adapter*.

Isto é, esta extensão do `Calendar` estará acessível a todos que quiserem, e quem tiver um objeto do tipo `Calendar`, só precisará fazer o *import* da função, e conseguirá acessá-la normalmente.

O *adapter* deixa de ter o `formatoBrasileiro`, pois o deslocamos integralmente ao novo arquivo. Com o atalho "Ctrl + Shift + F12", perceberemos que ele não é reconhecido no *adapter*. Vamos precisar importá-lo, algo que o Android Studio nos sugere automaticamente.

Para ajustar os *imports* usaremos "Ctrl + Alt + O". Nosso arquivo é importado como `extension.formataParaBrasileiro`. Ou seja, o programa está importando uma função diretamente, não sendo necessária uma classe para isto.

Esta é uma das diferenças entre um método e uma função, que **não depende necessariamente de uma classe para existir**. Podemos criá-la em arquivos e utilizá-la da maneira que quisermos. Este também é um dos paradigmas que o Kotlin dá suporte: o **paradigma funcional**.

Não somos obrigados a criar métodos para usar comportamentos de blocos de códigos. E é possível simplificarmos ainda mais, alterando `val dataFormatada` e retornando o `format` diretamente, deletando a chamada de `dataFormatada`, que era nossa variável:

```
fun Calendar.formataParaBrasileiro() : String{
    val formatoBrasileiro = "dd/MM/yyyy"
    val format = SimpleDateFormat(formatoBrasileiro)
    return format.format(this.time)
}
```

Assim, temos acesso ao `formataParaBrasileiro`, que poderemos usar em outro lugar caso queiramos, bastando apenas importarmos esta função. Vamos verificar se tudo está funcionando realmente?

Voltando ao nosso emulador, a diferença agora é que estamos fazendo com que o objeto `Calendar` faça a formatação, já que este é um comportamento comum na aplicação.

> **Atenção!** Este tipo de comportamento é **muito poderoso**, e permite que coloquemos ações em classes que não são nossas, o que é incomum no Java. Isso tende a ser perigoso dependendo da forma com que lidamos com isso. Se definirmos comportamentos não esperados em funções, por exemplo, e outro programador tiver que trabalhar com o código, ele pode acabar não conseguindo ou não entendendo o porquê daquilo.
>
> Se para você fizer muito sentido estender uma classe para um comportamento que é muito comum em sua aplicação, aí sim a *Extension function* é recomendada. Caso contrário, **evite o máximo possível**, pois você poderá estar piorando seu código ao invés de melhorá-lo.