18

Implementando o próprio Adapter - Parte 2

Transcrição

Continuando a implementação do `BaseAdapter`, vamos começar com as funções mais simples, a que devolve um item (`getItem`) e aquela que conta a quantidade de itens disponíveis no *adapter* (`getCount`).

Agora, precisaremos de uma coleção para representar as funções e seus elementos, e recebermos uma lista, a `ListaTransacoesAdapter`, via construtor.

Uma das abordagens iniciais no Kotlin consiste em utilizarmos o construtor primário, que vem logo antes da classe. Podemos informá-la de que queremos receber parâmetros via construtor. O IDE diz que é possível remover o construtor vazio, já que nada está sendo recebido - isto é, a existência do construtor já vem subentendida, se não colocamos nada.

Com este tipo de construtor, faz todo sentido colocarmos um parâmetro para recebermos nele, pois aí sim será diferente do construtor padrão, o qual não recebe nenhum parâmetro.

Agora que indicamos que queremos ter algo no construtor, precisaremos de uma lista de transações. E já que em nossa *Activity* utilizamos uma lista de *string*, a princípio, vamos indicar que receberemos uma de *strings* também. Mais adiante veremos uma maneira mais adequada de fazê-lo. Por ora, só para vermos como funciona, usaremos assim.

Quando recebemos este tipo de informação via parâmetro, não temos como acessar via os membros da classe, porque é preciso algum tipo de atributo acessível para todos, já que só é possível acessá-lo durante a construção da classe `transacoes: List<String>`, e não para usar durante as funções, pois ele não está sendo armazenado na classe.

Em outras palavras, é necessário colocarmos a lista de transações dentro de um atributo, para que seja acessível nas funções. Para isto, usaremos `val`, que receberá as transações. Para reforçar a ideia, o que acontece é o seguinte: no Kotlin, já que conhecemos o tipo das transições, consegue-se inferir no tipo.

Repetindo: já que não temos a intenção inicial de mudar o valor, faz todo sentido deixarmos `val`, garantindo assim a integridade do código e evitando bugs inesperados.

É um comportamento bastante comum quando usamos orientação a Objetos; queremos disponibilizá-los somente quando há necessidade, e portanto usaremos `private`:

```
class ListaTransacoesAdapter(transacoes: List<String>): BaseAdapter() {

    private val transacoes = transacoes

    //...
}
```

Então, sempre que formos utilizar algum tipo de atributo em uma classe, é a mesma prática usada no Java: não o deixaremos acessível, pois não faz sentido ser acessível para todos.

Como já implementamos o *adapter* anteriormente, sabemos que o parâmetro `p0` referencia uma posição, mas para quem nunca viu isto, não ficaria tão claro. Para melhorar a semântica, alteraremos por `posicao`, referente à nossa coleção.

Informaremos também que queremos retornar das transações uma de acordo com sua posição, ou seja, pelo método `get`. É um processo similar ao que ocorre no Java, com uma pequena diferença: aparece uma mensagem dizendo que podemos substituir o `get` por meio de uma chamada indexada.

Isto significa que se colocarmos "Alt + Enter" ali, e acatarmos esta sugestão, o próprio Android Studio diz que, já que estamos usando uma lista, no Kotlin existe a possibilidade de chamar aquela mesma função `get` no mesmo estilo do `Array`, ou seja, com colchetes.

A partir deles, ao usarmos coleções como no caso desta lista, conseguiremos buscar um elemento. É uma abordagem mais sucinta em relação ao que é feito com as *collections* no Java.

A partir daqui, podemos tirar o `TODO()` usando "Ctrl + Y". Veremos que está sendo devolvido um objeto denominado `Any`, que no Kotlin é como se fosse o `Object` do Java, uma Superclasse de todas as classes.

Quando devolvemos `Any`, pode-se devolver uma *string*, ou qualquer outro tipo. Já que estamos de fato lidando com *strings*, faz sentido modificarmos o `Any`, deixando-o mais restrito e evitando termos que fazer *cast* ou ações do gênero.

Alteraremos `getCount()` usando `size()`, porém, ocorre algum problema. Utilizaremos o atalho "Ctrl + Q" para verificarmos a documentação de algum código, neste caso da nossa `List<>`, que veremos que vem do pacote `kotlin.collections`.

Poderemos ver isto de outra maneira, por meio do autocompletar, também. Em vez de utilizarmos as coleções do Java, do `pacote.util`, usamos as do Kotlin, com diversas implementações feitas nesta linguagem, não diretamente do Java.

Pensando neste detalhe, teremos certas peculiaridades, como é o caso do `size()` que chamaremos por meio de uma função chamada `property()`, assunto que veremos futuramente. É como se fosse um atributo da classe, que está sendo acessível para nós. A implementação de *collections* do Kotlin nos permite acessarmos o `size` a partir de `property`.

Legal! Conseguimos implementar tanto a função que devolve o item quanto a que mostra a quantidade de itens, agora vamos implementar a função que leva o *id*, retornando um valor `0` pois, até o momento, não temos nenhuma informação que represente um *id*, portanto não faz sentido colocarmos um em nossos itens.

Já implementamos as funções mais objetivas quando lidamos com *adapters*. A mais complexa é a que cria a *view* que, ao analisarmos, veremos que possui variáveis que não nos dão muito significado.

Já vimos que `p0` refere-se a posição, `p1` representa a *view* que está sendo utilizada no momento de `getView`, e `p2` é *parent*, a *view* que está mantendo todas as outras do nosso *adapter*.

Para criarmos uma tela no `getView`, é necessário inflá-la, ou dar um `inflate`. Para isto, existe a famosa classe `LayoutInflater`, que permite a criação de uma tela a partir de outra, proveniente de nossos *resources*.

Inicialmente, é necessário algum contexto para nos basearmos nele, por isto utilizamos `from`, acrescentando o contexto na classe.

Para deixar claro, quando usamos Kotlin e enviamos estas informações via parâmetro no construtor, há uma boa prática indicada pela JetBrains que consiste em, quando se recebe mais de um parâmetro, ficarem em linhas diferentes.

E da mesma maneira como fizemos com as transações, para podermos deixar acessível a todos, criaremos um `context` privado, informando que ele receberá um por parâmetro.

Chamaremos a função `inflate()` e acessaremos o *resource* com `R`, chamando o `layout`, apertando "Alt + Enter" para fazermos a importação, chamando também `transacao_item`. O próximo parâmetro a ser enviado é o *parent* `ViewGroup`, e no último parâmetro indicaremos se queremos criar nossa *view* neste momento.

Neste caso, já que estamos usando este comportamento no *adapter*, esta responsabilidade será delegada a ele, ou seja, não será criada. Feito o `inflate()` da *view*, precisaremos devolvê-la para utilizá-la, e conseguirmos devolvê-la ao `getView`.

Uma vez que não a utilizaremos neste momento, podemos simplesmente acrescentarmos `return` diretamente, só para vermos o código funcionando. Neste momento, o código se encontra da seguinte forma:

```
class ListaTransacoesAdapter(transacoes: List<String>,
                             context: Context) : BaseAdapter() {

    private val transacoes = transacoes
    private val context = context

    override fun getView(posicao: Int, view: View?, parent: ViewGroup?): View {
        return LayoutInflater.from(context).inflate(R.layout.transacao_item, parent, false)
    }

    override fun getItem(posicao: Int): String {
        return transacoes[posicao]
    }

    override fun getItemId(p0: Int): Long {
        return 0
    }

    override fun getCount(): Int {
        return transacoes.size
    }
}
```

Para verificarmos se o layout desejado aparecerá, e se tudo isto foi computado corretamente, vamos devolver `LayoutInflater` diretamente. Mas não se preocupe! Mais para a frente pegaremos a *view* e a customizaremos de acordo com o que recebermos de valores ali dentro.

Vamos voltar à *Activity* e verificar se tudo está correto. Veremos que é preciso enviar o `context` para lá também:

```
lista_transacoes_listview.setAdapter(ListaTransacoesAdapter(transacoes, context: this))
```

Executaremos nossa app com "Alt + Shift + F10". O Android Studio consegue executar a aplicação sem nenhum problema, com o *adapter* que definimos, o layout `transacao_item`.

Em seguida, tentaremos deixar a nossa app com o mesmo aspecto que vimos inicialmente, com informações sobre valores de receitas e despesas, categoria, data e afins.