19

Criando Adapter personalizado

Embora tenhamos conseguido settar um adapter na nossa lista, precisamos chegar a um resultado similar ao que temos atualmente na App feita em Java:

Para isso, precisamos criar o nosso próprio Adapter que vai utilizar esse layout personalizado.

Criando o Adapter personalizado

Portanto, crie a classe `ListaTransacoesAdapter`, em seguida, faça a extensão da classe `BaseAdapter`. Com a herança realizada, peça para o AS implementar as funções que precisam ser sobrescritas usando o atalho **Alt + Enter**.

> O Android Studio usa como base a IDE IntelliJ IDEA (https://www.jetbrains.com/idea/), ou seja, você pode usar grande parte dos atalhos e recursos do IntelliJ no Android Studio também. Caso seja iniciante na ferramenta, temos também um curso de IntelliJ (https://cursos.alura.com.br/course/intellij-idea-truques-para-aumentar-sua-produtividade-em-projetos-java) que ensina dicas e truques para aumentar a produtividade em projetos Java que, por sinal, são técnicas que podem ser usadas no AS.

Repara que nesse instante surgiram as funções:

- **getView()**: cria a view
- **getItem()**: retorna o item pela posição
- **getItemId()**: retornar o id do item pela posição
- **getCount()**: retorna o tamanho da lista

Dentro de cada uma delas também surgiu a função `TODO()`. Como vimos, trata-se de uma função que quando é chamada, crasheia a App, ou seja, ela serve para "evitar" que a função seja executada caso não tenha sido implementada ainda. Em outras palavras, após implementar cada uma das funções, remova todas as funções `TODO()`.

Implementando as funções mais básicas

Para implementar as funções `getItem()` e `getCount()` é necessário a composição de uma lista, portanto, peça lista de `String` via construtor e atribua para um atributo da classe.

Então, dentro do `getItem()` devolva uma String por meio da posição e, no `getCount()` devolva o tamanho da lista.

A implementação da função `getItemId()` nesse nosso caso pode ser feita apenas devolvendo o valor `0`, pois não temos um campo id para representar transações até o momento.

Implementando a função que vai construir a view

Agora precisamos implementar a função que vai construir o container de cada uma das transações. Para isso, chame a classe a função `from()` da classe `LayoutInflater` de forma estática.

Entretanto, repara a função `from()` exige o envio de um `Context` , portanto, faça a composição de um context atribuindo o seu valor via construtor, da mesma forma como foi feito na lista de Strings para representar as transações.

Em seguida, chame a função `inflate()` e envie como argumento o layout `transacao_item` . Lembre-se de mandar tanto o `parent` (`ViewGroup`) como último argumento envie `false` para que a view seja criada pelo Adapter. Para finalizar a implementação, retorne a função `inflate()` como retorno da função `getView()` .

Depois de implementar todas as funções, crie uma instância da classe `ListaTransacoesAdapter` e envie como adapter na `ListView` . Repara que agora é necessário enviar a lista de Strings que representa as transações, como também, o contexto. Por fim, execute a App e veja se aparece o container das transações.