01

Criando o modelo para transação

Transcrição

Conseguimos criar uma lista para representar cada uma das transações, porém, se compararmos com o projeto feito em Java, veremos que neste temos valor, categoria, data e ícone. Em nossa aplicação, não há nenhum tipo de representação visual para estas informações.

Anteriormente, utilizamos `ArrayAdapter`, uma lista de *strings*, em que não é possível deixar diversos valores e acessá-los de maneira objetiva e fácil. Como fazer isso?

Usaremos uma estrutura capaz de armazenar estes valores, então criaremos classes para tal, uma classe modelo que represente nossa transação. Para isso, da mesma maneira como fazemos em Java, acessaremos `br.com.alura.financask.ui` no menu lateral e criaremos um pacote *model*.

Digitaremos "model", e o pacote é criado em "ui", sendo que ele deveria estar no mesmo nível do pacote raiz. Usaremos "F6", o recurso de "*move*", para movermos um pacote a outro.

Em seguida utilizaremos "Alt + Insert", selecionaremos "Kotlin File/Class" e escreveremos "Transacao", colocando-a com tipo "Class" e clicaremos em "OK".

Agora, acrescentaremos as informações de que precisamos, criando os atributos para a classe: um valor de tipo `BigDecimal`, que importaremos. O Kotlin informa que ele precisará ser inicializado - por que isto ocorre?

Como o Kotlin é uma linguagem de programação que visa muito as boas práticas, ele não deixará com que as variáveis sejam declaradas sem inicialização, sejam atributos ou variáveis locais.

Isto significa que toda vez que declararmos uma variável, independentemente do que ela seja, ela precisará ser inicializada. Uma das formas de fazê-lo é deixar `BigDecimal` com `ZERO`, porém, assim, faremos com que o valor seja modificado, sendo que na verdade queremos que um valor seja recebido em uma classe somente durante sua construção, sem que haja modificação.

Portanto, há a opção de deixarmos a variável como `val`, retirando a inicialização, enviando o comportamento de alguém inicializá-lo por meio do construtor, como fizemos com o *adapter*.

Em `ListaTransacoesAdapter`, recebíamos o valor no construtor, e ele mesmo inicializava o atributo. Isto é, receberemos um valor do tipo `BigDecimal`, que ficará responsável por atribuir e inicializar nosso atributo.

O código ficará assim:

```
package br.com.alura.financask.model

import java.math.BigDecimal

/**
 * Created by Alura on 22/09/2017.
 */
class Transacao (valor: BigDecimal){
```

```
    private val valor: BigDecimal = valor
}
```

Ao observarmos o *adapter* novamente, veremos que não colocávamos o tipo em `val`, pois algumas linhas acima já está bem claro que trata-se de uma *string*, assim como em `BigDecimal` já fica bem claro o tipo do parâmetro que estamos recebendo. Assim sendo, pode-se deixar o código assim:

```
private val valor = valor
```

Ao nos depararmos com este tipo de situação, se colocarmos `String` no código, nossa classe usará este atributo como sendo uma *string*, e às vezes este não é o comportamento esperado. Às vezes você quer deixar bem claro que trata-se de um `BigDecimal`.

Sinta-se à vontade para deixar tanto em cima quanto embaixo como `BigDecimal`, apenas lembrando-se de que é importante deixar o tipo explícito quando se quer garantir isto, e que quando você recebê-lo via parâmetro, isto não quebrará sua app com um comportamento inesperado.

Com isto, conseguimos implementar nosso atributo! Vamos à categoria, que poderemos colocar como uma *string*, um texto simples, e a data, que será uma API do próprio Java, o `Calendar`.

Notem que ao incluirmos o `Calendar`, o programa importou todos os pacotes de `util`, quando na verdade só estamos utilizando a API, então substituiremos `*` por `Calendar`.

Devemos prestar atenção nestes *imports*, pois muitas vezes eles não são necessários. Colocaremos os atributos:

```
package br.com.alura.financask.model

import java.math.BigDecimal
import java.util.Calendar

class Transacao (valor: BigDecimal,
                 categoria: String,
                 data: Calendar){

    private val valor: BigDecimal = valor
    private val categoria: String = categoria
    private val data: Calendar = data
}
```

Agora temos uma classe do tipo `Transacao`, e precisaremos dos objetos de mesmo tipo na nossa *Activity*. Em vez de usarmos uma lista de *strings*, usaremos uma de `Transacao`, portanto iremos instanciá-la, passando três valores:

```
val transacoes = listOf(Transacao(BigDecimal(20.5), categoria: "Comida", Calendar.getInstance()
```

Não precisaremos mais do `ArrayAdapter`, pois estaremos utilizando apenas o *adapter* que criamos, então poderemos deletá-lo. O programa aponta um problema, dizendo que precisa de uma lista de *strings*, porém estamos enviando uma lista de `Transacao`. Vamos alterar isto no *adapter*.

Usando o atalho "Alt + Enter" com o cursor sobre `transacoes`, sugere-se a modificação do parâmetro para o tipo que estamos colocando agora, que é o que queremos. Ao acessarmos `ListaTransacoesAdapter`, veremos que `List<String>` tornou-se `List<Transacao>`.

Porém, reparem que `transacoes[posicao]` deixou de ser compilado. Estávamos usando `Any`, mas tínhamos trocado-o por `String`. Se voltarmos ao primeiro, a compilação ocorre sem problemas, no entanto teremos que lidar com *cast*. Já que temos uma transação, deixaremos `Transacao`:

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

Tendo uma lista de transações, o que faremos para que nosso *adapter* consiga acessar as informações das transações, enviando-as para a tela do app?

No momento em que ela pega a *view*, acessaremos seus componentes e mandaremos as informações de algum objeto, e precisaremos trabalhar nessa função. Neste primeiro momento, acessaremos a *view* que está sendo criada, da mesma maneira como criávamos variáveis, acrescentando `.val` e apertando "Enter" em seguida, o que gerará um template para nós:

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

O Android Studio nos dá duas opções de criação de variável: a partir da expressão ou do retorno. No caso, queremos uma variável desta expressão, cujo nome será "viewCriada".

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

Reparem que sempre que utilizamos `.val`, existem alternativas para declaração de variáveis, sendo a primeira delas declararmos como `var`. Outra alternativa implica em se especificar isto de forma explícita de definindo-se o tipo.

Vamos marcá-la e apertar o "Enter" para ver o que acontece - o programa mostra o tipo inferido para nós, neste caso, uma *view* de tipo `View`. Não precisamos deixar isto à mostra, pois já está subentendido.

Após a criação desta *view*, como acessaremos os componentes contidos em `transacao_item`? Observando este layout, veremos que há `transacao_data`, `transacao_categoria`, `transacao_icone`, por exemplo. Usando "Ctrl + F4" para fecharmos a aba, chegaremos a eles por meio de `viewCriada`, a qual contém o layout.

Pode-se fazê-lo pelo `findViewById`, porém, como visto anteriormente, conseguiremos utilizar o `synthetic` em nossa *Activity*, que utiliza um layout e faz com que todos os seus componentes se tornem propriedades da *Activity*, como é o caso da `listView`.

Felizmente, o `synthetic` não está atrelado somente a uma *Activity*, estando também a objetos do tipo `View`, como a `viewCriada`. Se digitarmos `transacao_valor`, poderemos fazer o *import* com `synthetic`, utilizando o `main`, o que se

pode ver em:

```
import kotlinx.android.synthetic.main.transacao_item.view.*
```

Deste modo, disponibilizam-se os componentes a partir da *view*. Quando estão em uma *Activity*, um objeto de tipo `View` não é necessário, e quando estamos em outra entidade, que não é uma *Activity*, por exemplo um *adapter*, conseguiremos utilizar o `synthetic` com `import` .

Agora que temos um `TextView` , poderemos setar os valores, para dentro do qual passaremos o valor da transação, buscando o item de uma lista com colchetes:

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

    val transacao = transacoes[posicao]

    viewCriada.transacao_valor.setText(transacao)


    return viewCriada
}
```

Declaramos todos os atributos de maneira que ninguém consegue acessá-los, portanto teremos que alterar isto, assim como fazemos em Java, criando uma função que irá devolver o valor, um `getValor()` . Devolveremos `BigDecimal` retornando o valor em si:

```
fun getValor() : BigDecimal {
    return valor
}
```

Voltando ao *adapter*, teremos:

```
viewCriada.transacao_valor.setText(transacao.getValor())
```

Em que `setText` não é compilado, pois espera-se um `CharSequence` , e estamos enviando um `BigDecimal` , ou seja, precisaremos fazer com que este valor seja uma *string*, um `toString` :

```
viewCriada.transacao_valor.setText(transacao.getValor().toString())
```

Assim, a compilação ocorre com sucesso e nosso código funciona corretamente! Vamos executar a aplicação e verificar seu comportamento através do emulador. Os valores são mostrados!