01

Utilizando classes enum

Transcrição

Neste momento, alcançamos a representação visual que indica cada uma das transações em nossa lista. Abrindo a app base no emulador, aquela desenvolvida em Java, perceberemos alguns detalhes que ainda não foram implementados.

Nela, notamos logo de cara que uma transação de receita está na cor azul e se localiza ao lado do ícone com seta para cima, enquanto a transação de despesa é identificada pela cor vermelha e um ícone com seta para baixo.

Ao incluirmos uma nova receita de "20" como valor, clicando em "Adiciona receita", e definindo-a como "Presente", ela fica com o visual correspondente à identificação de receitas. O mesmo acontece no caso de uma despesa de valor "15" e "Lazer", que receberá um ícone vermelho.

Desta forma distinguimos uma transação da outra, algo que ainda não acontece na nossa app atualmente, pois não há uma maneira fácil de diferenciar a despesa da receita.

Com "Ctrl + N" buscaremos `Transacao`, em que há as *properties* `valor`, `categoria` e `data`. A princípio, não há nada que demonstre a distinção das transações, portanto adicionaremos mais uma propriedade, com intuito de indicar isto.

Uma *property* adequada seria a variável booleana `val`, uma vez que não pretendemos alterar seu valor. Caso queiramos lidar com uma receita, enviaremos o parâmetro `receita` como `true`, e se tratar de uma despesa, o enviaremos como `false`:

```
class Transacao (val valor: BigDecimal,
                 val categoria: String,
                 val receita: Boolean,
                 val data: Calendar)
```

Porém, este tipo de abordagem possui certas limitações, o campo `receitas` pode não funcionar, pois ele está atrelado a apenas dois tipos de situação e, pior, aqui só estamos trabalhando com a parte de receita, então, como saberemos que é uma despesa, ou qualquer outra transação? Como teremos este tipo de informação sendo `false`, por exemplo?

Para usarmos um valor bem identificado, extensível e com grande significado, utilizaremos um recurso bem comum no Java: os **Enums**. Para criá-los em Kotlin, acessaremos nosso projeto, em "java > br.com.alura.financask > model", e criaremos um novo arquivo com "Alt + Insert", selecionando "New Kotlin File/Class".

Definiremos o novo arquivo como sendo de tipo "***Enum class***", a partir do qual criaremos todas as outras opções de transações existentes, com a possibilidade de acrescentarmos quaisquer outras. O nome será "Tipo", que é o que precisamos.

Agora, basta definirmos os tipos das transações do nosso projeto. Assim, teremos uma classe, no caso um Enum, representando todos os tipos de transações necessárias. Se surgir outra transação posteriormente, como `NOVA` ou `INDEFINIDA`, conseguiremos utilizá-la sem problemas, acrescentando-a no código abaixo:

```
enum class Tipo {
    RECEITA, DESPESA
}
```

Para usá-la na classe `Transacao`, é preciso substituir `receita` por `tipo` e, em vez do `Boolean`, que possui apenas `true` ou `false` e não nos contextualiza tanto, usaremos `Tipo`, nosso Enum que é importado pelo IntelliJ a partir de "Enter":

```
class Transacao (val valor: BigDecimal,
                 val categoria: String,
                 val tipo: Tipo,
                 val data: Calendar)
```

Para indicar esta informação no momento da instância, assim como fazemos as transações, acessaremos "activity" no menu lateral e, em "ListaTransacoesActivity", veremos que o projeto foi quebrado, pois agora é necessário enviar informações sobre o tipo.

> Para facilitar a navegação entre tabs no Android Studio, pode-se usar "Alt" com a seta para o lado direito ou esquerdo.

Em `Transacao`, `Tipo` aparece como o terceiro parâmetro do construtor, então faremos o mesmo em `ListaTransacoesActivity`, chamando o `Tipo` (o Enum) e incluindo `DESPESA`. O Android Studio pede para que haja importação, portanto apertaremos "Enter".

Definiremos "Economia" com `Tipo RECEITA` e obteremos:

```
val transacoes = listOf(Transacao(BigDecimal(20.5),
        categoria: "Comida", Tipo.DESPESA, Calendar.getInstance()),
        Transacao(BigDecimal(100.0), categoria: "Economia", Tipo.RECEITA, Calendar.getInstance(
```

Com isso incluímos nossas transações e identificamos cada uma delas com seu tipo correspondente. No entanto, a instância da transação está ficando bem extensa, e toda vez que precisarmos fazê-la, teremos que incluir estas informações também. Será que não existe uma maneira mais resumida de fazermos isso?

Estamos colocando todas as datas como `Calendar.getInstance()`, pois não definimos uma data específica... Será que não conseguimos colocar estes valores de forma mais objetiva?

No Kotlin, é possível fazer com que os parâmetros do construtor tenham valores padrões, possibilitando que sempre que alguém for utilizar a classe `Transacao` sem enviar uma data, utilizemos este valor como padrão.

Para isto, incluiremos informações à *property*, aquilo que esperamos como valor padrão - o `Calendar.getInstance()` - toda vez que alguém for instanciar a transação e quiser usar a data do dia corrente.

```
class Transacao (val valor: BigDecimal,
                 val categoria: String,
                 val tipo: Tipo,
                 val data: Calendar = Calendar.getInstance())
```

Voltaremos à *Activity* para remover a data e verificar a compilação:

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

```
        Transacao(BigDecimal(100.0), categoria: "Economia", Tipo.RECEITA))
```

Isto resume um pouco mais a nossa instância! Vamos testar a aplicação para ver se ela funciona? O Android Studio e o emulador conseguem rodá-la sem problemas, isto é, as informações sobre a data são enviadas corretamente, mostrando sempre a atual.

A princípio, não queremos alterar o valor das *properties*, mantendo-se o valor padrão. Além disso, pode-se fazer o mesmo para outros campos, não necessariamente para apenas um. Se não houver nenhuma categoria definida, por exemplo, e nenhuma informação for enviada, ele será mantido como indefinido por padrão:

```
class Transacao (val valor: BigDecimal,
                 val categoria: String = "Indefinida",
                 val tipo: Tipo,
                 val data: Calendar = Calendar.getInstance())
```

Da mesma maneira como fizemos com a data, removeremos `"Comida"` (a categoria), deixando o código como se vê abaixo:

```
val transacoes = listOf(Transacao(BigDecimal(20.5),
        Tipo.DESPESA),
        Transacao(BigDecimal(100.0), categoria: "Economia", Tipo.RECEITA))
```

Porém, o código não está sendo compilado! Na classe `Transacao`, os parâmetros que estamos enviando são um `valor`, que é um `BigDecimal`, o `categoria`, que é uma `String` e agora possui valor *default* (padrão) `"Indefinida"`, um `tipo` e uma `data`, também com valor padrão (`Calendar.getInstance()`).

Ao acessarmos a *Activity*, veremos que inicialmente enviamos `BigDecimal` para representar o `valor`. Neste momento, enviamos um `Tipo`, sendo que na verdade é esperada uma categoria, o valor `String`, por mais que tenhamos definido o valor padrão.

Como resolveremos este detalhe?

Com "Ctrl + Shift" e as setas para cima e para baixo, moveremos a `categoria`, deixando-a abaixo de `tipo`, assim:

```
class Transacao (val valor: BigDecimal,
                 val tipo: Tipo,
                 val categoria: String = "Indefinida",
                 val data: Calendar = Calendar.getInstance())
```

Voltaremos à *Activity*, em que houve compilação conforme esperado. Mas tem um detalhe! Agora, o `tipo: "Economia", Tipo.RECEITA` é que não compila! Significa que não conseguimos enviar a categoria no segundo parâmetro. Tal como fizemos anteriormente, o projeto está sendo quebrado, e a instância, da maneira como tinha sido feita, deixou de funcionar.

Uma abordagem válida seria inverter os parâmetros, deixando `Transacao(BigDecimal(100.0), Tipo.RECEITA, categoria: "Economia"))`. Porém, **tendemos a correr riscos com este tipo de comportamento**. Este projeto com que estamos trabalhando é pequeno, e permite testes e alterações. Se fosse um projeto grande, com muitas instâncias de transações, poderíamos quebrar o projeto atingindo um ponto cujas manutenções seriam inviáveis.

Então, este tipo de abordagem **não é desejada. Mudar a ordem de parâmetros do construtor não é recomendado!**

De que forma faremos com que as classes possam ter maior quantidade de *properties*, evitando situações nas quais é possível incluir valores padrões, sem que tenhamos que alterá-los? Veremos no próximo vídeo!