05

Entendendo as properties

Transcrição

Implementamos o *adapter* para que sejam mostrados os valores das transações da lista a partir de uma delas em `getView()`, a qual chama a função `getValor()`. Acessando-a por meio do "Ctrl + B", vê-se que ali se faz o encapsulamento do atributo `valor`.

Esta é a maneira como acessamos os atributos em Java, com seus **métodos de acesso**. No caso, fazemos funções de acesso, com `getValor()`. Lembrando que o Kotlin tende a deixar o código o mais conciso possível, em alguns momentos não é preciso fazer códigos comuns ao Java para obtermos um comportamento similar.

É o caso de leitura e escrita em um atributo - não é necessário colocarmos `getValor()`, podemos apagá-lo e expor o atributo. Isto é possível no Kotlin porque não estamos expondo-o neste momento, e sim uma *property*.

Significa que, em vez de ser um atributo acessível e modificável por todos, ao deixarmos a *property* exposta, todos que a usarão farão isto implicitamente, com seus *gets* e *sets*. Não é possível fazê-lo sem passar por eles. `Transacao.kt` ficará da seguinte maneira:

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

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

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

}
```

Enquanto em `ListaTransacoesAdapter`, teremos

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

É comum isso acarretar em diversas dúvidas: não se acessa o atributo diretamente? Isso não é perigoso, já que qualquer um pode modificá-lo ou manipulá-lo da maneira que quiser? Entenderemos que acessamos o *get* e fazemos as modificações no *set*.

Para compreendermos este comportamento das *properties* utilizaremos uma ferramenta do Kotlin acessando "Tools > Kotlin > Kotlin REPL", que abre um terminal interativo para escrevermos um código que será lido, interpretado e impresso, voltando depois ao *loop*.

"REPL" indica justamente "*Read-Eval-Print-Loop*", e para testarmos seu funcionamento, colocaremos um código bem simples, um `println`, que faz a impressão do console:

```
println("Bem vindo ao REPL")
```

Faremos a execução com "Ctrl + Enter", a partir do qual o REPL irá ler, interpretar e imprimir. Começaremos a testá-lo para mostrar que o que fizemos na `Transacao` é uma *property*, e não um atributo.

Criaremos uma classe `Transacao` para deixar o mais minificado possível, e colocaremos apenas uma transação que recebe um valor de `BigDecimal`. Veremos que ele é importado e repetiremos o procedimento feito anteriormente na classe, que recebe um valor em seu construtor.

Faremos sua instância, e o programa nos dá duas opções: do REPL e a do *model*. Já que o primeiro é uma ferramenta em desenvolvimento e ainda apresenta *bugs* inesperados, usaremos a outra opção, acrescentando-a em um objeto `val`. Faremos a sua impressão também:

```
import java.math.BigDecimal

class Transacao(valor: BigDecimal){
    val valor: BigDecimal = valor
}

val transacao = Transacao(BigDecimal(100.0))
println(transacao.valor)
```

Com "Ctrl + Enter", obteremos a impressão do valor `100`, algo esperado. Para voltarmos ao código do REPL que executamos antes, apertaremos a tecla com a seta para cima. Se tentarmos modificar a *property* da maneira em que está atualmente, incluindo a linha `transacao.valor = java.math.BigDecimal(20.5)`, o programa não deixa!

Isto ocorre porque o `val` possui o mesmo comportamento de uma constante, então, desta maneira estamos apenas lendo-o, sem conseguirmos modificá-lo. O que acontece quando temos um atributo dentro de uma classe? Às vezes queremos permitir que ele seja escrito, que alguém consiga modificar seu valor da maneira que quiser. Utilizamos os *getters* e *setters* para isso.

Vamos deixá-lo mutável trocando `val` por `var`. Executando o código, obtém-se a impressão de `20.5`. Ou seja, o atributo está sendo modificado diretamente, então devemos colocar um *set* no meio.

Para nos certificarmos disto, voltaremos no código usando a seta para cima e modificaremos o *set* deste valor digitando `set`, o que nos dá algumas opções de templates, dentre os quais escolheremos `set(value) {...}`, já que ele permite que coloquemos um bloco de código em seu processo.

Incluiremos uma mensagem para verificarmos se o *set* está sendo chamado no momento em que tentamos fazer esta atribuição:

```
class Transacao(valor: BigDecimal){
    var valor: BigDecimal = valor
    set(value) {
        println("Acessando a property valor")
    }
}
```

Com "Ctrl + Enter" mais uma vez, receberemos `Acessando a property valor100`. De fato, quando colocamos a atribuição `transacao.valor = java.math.BigDecimal(20.5)`, estamos chamando o *set*.

Retornando ao código anterior, poderemos retirar a parte de leitura (`println(transacao.valor)`) e verificar se isto é verdade, usando "Ctrl + Enter". Lembrando que colocamos o valor `100.0` logo no início e, anteriormente, quando colocamos `valor` para ser lido, deveria ser `20.5` , mas na verdade o que apareceu foi `100` .

Isso porque modificamos o *set*! Não estamos mais colocando `value` diretamente em nossa *property*. De que maneira podemos modificar o *set*, fazer algum processo nele e colocar o valor que recebemos como parâmetro na propriedade?

Inicialmente, poderíamos colocar o `valor` , nossa *property*, e fazer sua atribuição novamente com o `value` que está vindo em `set(value)` . Vamos tentar executar o código e ver o que acontece.

Com "Ctrl + Enter", vê-se a impressão de diversas mensagens:

```
Acessando a property valorAcessando a property valorAcessando a property valorAcessando a propei
```

Por que isto está acontecendo? Para entendermos melhor, voltaremos ao código anterior com a seta para cima, apagaremos `println("Acessando a property valor")` e executaremos mais uma vez. Usando "Ctrl + Enter", receberemos `java.lang.StackOverflowError` , um erro de *StackOverflow*.

Percebam que o próprio REPL nos indica um dos motivos disso ocorrer: em `valor` , há uma **chamada recursiva**. Isto é, quando tentamos acessar um `set()` de uma *property* e, internamente, seu valor (`valor`), na verdade tentamos refazer uma atribuição ao *setter* da propriedade.

Isto é, em `valor` , chamamos o *set* novamente, e quando alcança-se `set(value)` , ele é chamado de novo, e por aí vai, resultando em uma chamada sem fim. Neste momento, portanto, não estamos conseguindo modificar o valor da *property*.

Pode ser que queiramos alterar o *set*, fazer um processo nele para depois colocarmos o valor do atributo diretamente, na *property*. Para tal, o Kotlin nos fornece o recurso conhecido como **Backing Fields**, a partir do qual é possível acessar o valor interno da *property*.

O código, então, ficará desta forma:

```
class Transacao(valor: BigDecimal){
    var valor: BigDecimal = valor
    set(value) {
       field = value
    }
}
```

Rodando a aplicação com "Ctrl + Enter", o que se imprime é o `20.5` . Então, por debaixo dos panos, por padrão, há este comportamento do *set*, que recebe `value` e coloca-o diretamente na nossa *property*, algo que o Kotlin faz automaticamente. E como vemos, não precisaremos mais dos *gets* e *sets*.

Inclusive, se tentarmos modificar os *gets*, incluindo-os no código, modificando-os e acrescentando uma mensagem, vejam que se pede o retorno de um valor, o que faremos com `return` colocando um valor compatível com a *property*:

```
class Transacao(valor: BigDecimal){
    var valor: BigDecimal = valor
    set(value) {
```

```
        field = value
    }
}
get() {
    println("acessando o get da property valor")
    return BigDecimal.ZERO
}
```

Executando-se a app, notaremos que o *get* da *property* está sendo executado, retornando-se o valor `0`, e não o valor da *property*, de forma que sobrescrevemos sua forma de leitura. Assim, nunca mais teremos seu valor, aquele que foi acessado antes. Para isto, poderemos utilizar o Backing Field:

```
get() {
    println("acessando o get da property valor")
    return field
}
```

Vamos ver o que acontece? É retornado `acessando o get da property valor20.5`. É desta maneira que o Kotlin funciona quando utilizamos os atributos da classe, sendo que na verdade estamos usando as *properties*.

Observaremos que o REPL começa a ficar um pouco mais lento, porque às vezes a ferramenta apresenta comportamentos inesperados. Se isto acontecer, pode-se fechar a tela e acessar novamente "Tools > Kotlin > Kotlin REPL", selecionando o módulo da "app". Porém, deste modo, é preciso declarar novamente a classe com que estávamos trabalhando, pois todo o histórico do que foi executado se perde.

Neste momento, não mexeremos mais com *sets* e *gets* e veremos outras situações comuns, como quando se escreve o atributo apenas uma vez, permitindo apenas sua leitura para os nossos clientes.

Ou seja, neste cenário, se queremos manter a leitura de nosso atributo, deixaremos com `val`, que nos protege de alguém externo ou interno de modificar o valor da *property*. Portanto, se declararmos uma função na classe que altera o valor, e queremos que ele seja alterado internamente também, não é possível.

Assim, não conseguiremos reassinar o valor, pois trata-se de um comportamento não esperado. No entanto, como faremos quando a *property* é um `var`, mutável por dentro, mas não por fora? Poderemos usar `private set`:

```
class Transacao(valor: BigDecimal){
    var valor: BigDecimal = valor
    private set
}
    fun alteraValor(){
        valor = BigDecimal(10.0)
    }
```

Se queremos manter apenas leitura, usaremos *property* como `val`, e se queremos que ela seja mutável internamente, usaremos `var` e modificaremos o `private` com `set`. Se queremos que todos tenham acesso, basta tirarmos tudo, deixando simplesmente:

```
class Transacao(valor: BigDecimal){
    var valor: BigDecimal = valor
}
```

```
val transacao = Transacao(BigDecimal(100.0))
transacao.valor = BigDecimal(20.5)
println(transacao.valor)
```

O importante é entendermos que **na verdade estamos lidando com *sets* e *gets*, e não com acesso direto**. Uma última questão sobre as *properties*: em vez de termos que colocá-las no corpo da classe, há uma forma mais enxuta de fazer isto.

Para que os valores recebidos no construtor primário sejam de fato uma *property*, o código ficará assim, inclusive sem corpo (as chaves):

```
class Transacao(var valor: BigDecimal)
```

Podemos declarar uma *property* que todos leem e escrevem, bem como declarar aquela que é escrita apenas uma vez, sem que ninguém possa modificar seu valor.

Continuaremos modificando o código para deixá-lo o mais consistente possível. Até já!