04

Sobrecarga de construtores

Transcrição

Outra alternativa para resolvermos a situação do vídeo anterior é uma **sobrecarga de construtores**, em que, além dos construtores para instanciar a transação, criaremos outro construtor, que irá receber outros parâmetros e realizar a transação.

Para obtermos outro construtor em nossa classe, pode-se abrir seu corpo por meio de colchetes, chamando a *keyword* `constructor`, que indicará ao Kotlin sobre a existência de um **construtor primário**, e que queremos incluir um novo, o **secundário**, o qual se localizará dentro do corpo da classe.

Com este segundo construtor, indicaremos os parâmetros a serem recebidos. Se queremos que ele atenda ao tipo de necessidade na qual enviamos um valor e receberemos um `Tipo`, deixaremos o código assim:

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

    constructor(valor: BigDecimal, tipo: Tipo)
}
```

Entretanto, o construtor secundário não está sendo compilado! O programa informa que é preciso chamar o construtor primário também. Ou seja, **toda vez que chamarmos um construtor secundário, caso haja um primário declarado, com as devidas *properties*, ele deverá ser chamado também**, independente de outros construtores que estiverem na classe.

Para isto, utilizaremos o operador dois pontos (`:`), o mesmo que usamos para criar uma extensão, junto à palavra reservada "**this**", representação do construtor primário.

Quando o utilizamos, indica-se que precisaremos informar alguns valores obrigatórios, como o **valor** e o **tipo** - os demais são opcionais:

```
constructor(valor: BigDecimal, tipo: Tipo) : this(valor, tipo)
```

No entanto, ao tentarmos enviar o `tipo`, ocorre um erro, pois tenta-se fazer uma chamada recursiva do próprio construtor, que atende ao mesmo parâmetro, sendo que na verdade é necessário enviar um construtor compatível com o primário.

Isto é, precisaremos enviar uma categoria! Já que iremos permitir este tipo de comportamento por meio de construtor secundário, deixaremos o valor `"Indefinida"` na parte do construtor secundário, sem deixar o valor *default* na parte de cima:

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

```
    constructor(valor: BigDecimal, tipo: Tipo) : this(valor, "Indefinida", tipo)
}
```

Esta abordagem permitirá que todos que forem chamar o construtor com valor e tipo não precisarão enviar a parte da categoria. Vamos apenas corrigir o envio da categoria na *Activity*:

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

Assim, conseguiremos fazer com que nossa classe tenha mais de um construtor, resultando em uma sobrecarga de construtores, o que permite chamar tanto desta forma quanto pelo construtor primário.

Ao chamarmos `valor` e `tipo`, de fato há um valor padrão, `"Indefinida"`. Quando chamamos um valor, `BigDecimal`, e também um tipo `DESPESA`, como resolveremos o fato de termos um construtor que está recebendo um valor, um tipo, e agora uma data, com `Calendar.getInstance`, uma vez que em `Transacao` não há nenhum construtor que atenda a isso?

Poderemos repetir aquilo que fizemos para receber o `valor` e o `tipo`, isto é, um construtor secundário com o acréscimo da data, chamando o construtor primário:

```
constructor(valor: BigDecimal, tipo: Tipo) : this(valor, "Indefinida", tipo)

constructor(valor: BigDecimal, tipo: Tipo, data: Calendar) : this(valor, "Indefinida", tipo, dat
```

Por mais que tenhamos atingido este tipo de comportamento, acabamos deixando o código muito grande, e qualquer outro desenvolvedor que porventura tiver que trabalhar nisto terá dificuldades de compreensão, então não é uma abordagem tão adequada.

Para mantermos seu comportamento e o melhorarmos visualmente, permitindo a chamada das transações de diversas formas sem nos preocuparmos com ordens e problemas de compatibilidade, uma alternativa é a *feature* conhecida por *Named parameter*. A partir disto o Kotlin permite indicar que o `BigDecimal` representa a *property* `valor`, bem como `Tipo` é um `tipo`, e o `Calendar` representa uma `data`:

```
val transacoes = listOf(Transacao(valor = BigDecimal(20.5),
        tipo = Tipo.DESPESA, data = Calendar.getInstance()),
        Transacao(BigDecimal(100.0), "Economia", Tipo.RECEITA))
```

Ao chamarmos os construtores desta forma, indicamos o que cada um dos nossos valores representa, sem a necessidade de nos preocuparmos com ordens, como poderemos confirmar se deixarmos o código assim:

```
val transacoes = listOf(Transacao(
    tipo = Tipo.DESPESA, data = Calendar.getInstance(), valor = BigDecimal(20.5)),
    Transacao(BigDecimal(100.0), "Economia", Tipo.RECEITA))
```

Acessando a `Transacao` e deletando os construtores, deixando o valor padrão como `"Indefinida"`, o código mantém sua compilação normalmente.

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

Voltaremos à `ListaTransacoesActivity` , em que é possível indicarmos o que cada um dos demais campos significa, lembrando que a ordem entre eles não importa **quando utilizamos os *Named parameters***:

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

Vamos testar o código para ver se tudo funciona adequadamente? Com "Alt + Shift + F10", veremos que a aplicação roda corretamente, mantendo-se `"Indefinida"` na primeira transação, pois não mudamos nada em relação à categoria, então o valor padrão está sendo utilizado.

Conseguiremos fazer este tipo de chamada a partir de funções também, se houver alguma com parâmetro, e informaremos o que queremos indicar. Continuaremos no próximo vídeo!