06

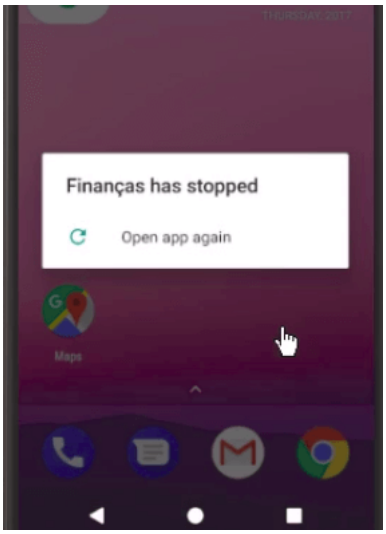
Lidando com Exceptions

Transcrição

Por mais que tenhamos feito com que o usuário consiga adicionar transações de receita na *app*, há mais um detalhe importante que precisamos dar uma olhada, e entender alguns casos excepcionais que acontecem quando desenvolvemos esse tipo de solução.

Em nossa *app*, temos o painel de resumo que está zerado, e um espaço embaixo para a lista de transações. Temos também o botão verde que podemos ter acesso a outros dois botões de função da *app*. Ao clicarmos em "Adicionar receita", vemos um *Dialog* onde contém os campos de `valor`, `data` e `categoria`.

No campo `valor`, damos a possibilidade do usuário inserir uma informação *vazia*, e o que isso pode agravar na *app*?



Ao tentar adicionar uma nova transação com o campo `valor` vazio, a aplicação quebra, como mostra na imagem acima. Vamos entender o por que ela quebrou?

No Android Studio, clicaremos em **Logcat** que se encontra na parte inferior da aplicação. Ao clicar nessa janela de log, vemos que foi lançado uma *NumberFormatException* na variável `valor`, onde é feito a conversão do valor que esta em texto, ou seja, ele tentou converter um valor **vazio** para um BigDecimal.

Em outras palavras, quando estamos lidando com esse tipo de solução que realiza **conversões**, existem casos em que precisamos nos atentar com essas situações. Portanto, faremos com que o código seja capaz de tratar esse tipo de situação por si só, informando o usuário, ou tomar alguma ação que precisa ser tomada, a fim de evitar que a *app* quebre. Podemos fechar o *Logcat*, para darmos início à implementação.

Como podemos evitar esse tipo de problema? Bom, da mesma maneira que fazemos no Java, podemos implementar um `try catch()`. Dentro do `try`, colocaremos todo o código que queremos "tentar" executar.

```
{ dialogInterface, i ->
    val valorEmTexto = viewCriada
            .form_transacao_valor.text.toString()
    val dataEmTexto = viewCriada
            .form_transacao_data.text.toString()
    val categoriaEmTexto = viewCriada
```

```
            .form_transacao_categoria.selectedItem.toString()

    try{
        val valor = BigDecimal(valorEmTexto)
    }
```

Quando tentamos executar algum código, queremos pegar algum erro, por isso colocaremos um `catch()`, e então diremos qual o tipo de erro que queremos pegar. O erro é uma *exception* e o seu tipo é um `NumberFormatException`.

```
try{
    val valor = BigDecimal(valorEmTexto)
} catch(exception: NumberFormatException) {

}
```

Perceba que será necessário criar o `valor` antes do escopo do `try`, pois faz mais sentido a transação ser criada antes de ser tratada. O que podemos fazer, é criar uma variável `valor` e dizer que seu valor inicial a princípio, será um *BigDecimal.ZERO*.

```
var valor = BigDecimal.ZERO
try{
    valor = BigDecimal(valorEmTexto)
} catch(exception: NumberFormatException) {

}
```

Esse código não tem casos em que ele pode quebrar por questões de conversão. Tentaremos realizar a conversão em `valor` e, caso dê certo, será atribuído para o valor que o usuário mandou. Dessa maneira conseguimos garantir que o código não quebre. Veja que manteremos o valor `ZERO`, e é algo não esperado pelo usuário. Porém somos capazes de informar isso para ele, através de um `Toast`:

```
var valor = BigDecimal.ZERO
try{
    valor = BigDecimal(valorEmTexto)
} catch(exception: NumberFormatException) {
    Toast.makeText(context: this,
            "Falha na conversão de valor",
            Toast.LENGTH_LONG)
            .show()
}
```

Desta maneira, conseguimos avisar para o usuário que teve um problema ao mandar uma informação sem nada.

Vamos executar a *app* novamente e tentar adicionar uma nova transação sem nenhum valor, e ver o que acontece.

A transação foi adicionada, porém é mostrado a mensagem para o usuário "Falha na conversão de valor". Então, por mais que tenhamos adicionado, estamos conseguindo passar uma informação para o usuário. Esse processo evita que o usuário fique sem saber porque a aplicação quebrou.

No Kotlin, temos alguns recursos novos quando lidamos com o `try catch` . Por exemplo, veja que estamos criando uma variável antes do escopo do `try catch` , sendo que estamos utilizando essa variável apenas para no caso de houver falhas.

No `try catch` , temos a capacidade de utilizar o *if expression* para trazer valores condicionais. O `try catch` pode nos devolver um `valor` , e caso dê certo, ele deve retornar um `BigDecimal` com o valor convertido baseando-se no `valorEmTexto` que está sendo recebido pelo usuário. Caso surja algum problema, ele irá devolver um `BigDecimal` com o valor zero.

```
var valor = try{
    BigDecimal(valorEmTexto)
} catch(exception: NumberFormatException) {
    Toast.makeText(context: this,
            text: "Falha na conversão de valor",
            Toast.LENGTH_LONG)
            .show()
    BigDecimal.ZERO
}
```

Ao executar o código novamente, a *app* apresenta o mesmo resultado. Agora não é mais necessário criar uma variável antes para poder entender que ela está sendo modificada dentro do `try` . Como vimos, ocorreu esse problema na ação simples de adicionar uma nova transação sem valor. Com a data isso já não acontece, pois o usuário é obrigado a selecionar um dia no calendário.

Entretanto, existe um detalhe na `data` . Se repararmos bem, o `parse` também lança uma *exception*. Ao lembrarmos do Java, nós temos algumas certas peculiaridade ao tratar de *exceptions*: as checked e unchecked. O `ParseException` é

uma **checked**, e o Kotlin não nos avisou sobre isso.

Estamos tendo um risco de uma *exception* que é **checked**, e não estamos tendo ciência disso. Nós não estamos tendo esse problema, porque temos a certeza de que a data está indo no formato que esperamos. É muito comum no dia a dia, receber valores que não esperamos em outros formatos, e quando ocorre a conversão, acaba falhando.

Isso significa que no Kotlin, temos uma abordagem de que não somos obrigados a implementar o `try catch` pra tratar qualquer *exception*, por mais que ela seja **checked**. A minha opinião para esse ponto é que a parte boa de saber que estamos lidando com uma *exception checked*, é que já iremos tomar os cuidados em uma coisa que pode ser comum em dar problema. A parte ruim é que sempre devemos colocar o `try catch` , e aumentando a complexidade de leitura do código. Há outras linguagens que não nos obrigam a fazer o tratamento de *exceptions*, mas isso não quer dizer que são linguagens ruins.

> No Kotlin, nenhuma *exception* que possamos receber, por mais que seja do Java, ela será **unchecked**.

Então, a ideia é que se acontecer algum problema, temos que pensar e tratá-lo, ou seja, não temos como ter uma ação prévia disso, sendo essa uma das diferenças. É claro, existem diversas maneiras de contornarmos essas situações, e uma delas por exemplo, é criar testes para ver se realmente o código está funcionando em diversas situações que esperamos no dia-a-dia.