01

Adicionando cores na receita e despesa

Transcrição

Atualmente, conseguimos adicionar as informações de total de receitas, despesas e a diferença entre elas no `ResumoView`. Porém, ao darmos uma atenção maior ao aspecto visual em relação à *app* base, vemos que o campo da **receita** possui uma cor que realmente representa uma receita, assim como a **despesa** está com uma cor que força a ideia de uma despesa. O mesmo acontece com o total.

Observaremos o comportamento das cores em relação a quantidade de transações que vamos colocando para alterar o que é necessário no código. Em outras palavras, vamos modificar o aspecto visual para que fique similar à App.

Para o primeiro teste, adicionaremos uma *receita* qualquer, no valor de `R$ 100,00`

Repare que a cor da receita foi mantida, a fim de reforçar a ideia de que isso é uma receita, e a cor da despesa continua a ser na cor vermelha. A cor do total, como podemos ver, é a mesma cor da receita, e isso acontece para indicar que estamos *ganhando* na questão líquida.

Adicionaremos uma *despesa* de `R$ 50,00`, por exemplo, para ver o que acontece.

Ele ainda mantém a cor da receita, a da despesa, e também a do total. Só que o que acontece se fizermos com que as despesas sejam maiores que a receita?

A nova despesa de teste, será de `R$ 100,00`.

Como podemos ver, a despesa é maior que a receita. Isso fez com que o total pegasse a cor da despesa, e isso nos força a pensar que estamos em "dívida" ou perdendo dinheiro.

É justamente esse aspecto visual que precisamos implementar em nossa *app*.

Na *activity* `ListaTransacoesActivity` , podemos perceber que temos mais **despesas** do que receitas.

```
private fun trasacoesDeExemplo().List<Transacao> {
    return listOf(Transacao(
        tipo = Tipo.DESPESA,
        categoria = "almoço de final de semana",
        data = Calendar.getInstance(),
        valor = BigDecimal(val: 20.5),
        Transacao(valor = BigDecimal(val: 100.0),
            tipo = Tipo.RECEITA,
            categoria = "Economia"),
        Transacao(valor = BigDecimal(val: 200.0),
            tipo = Tipo.DESPESA),
        Transacao(valor = BigDecimal(val: 500.0),
            categoria = "Prêmio",
            tipo = Tipo.RECEITA)
    )
}
```

Faremos uma modificação na transação de despesa, colocando um valor superior.

```
        Transacao(valor = BigDecimal(val: 700.0),
            tipo = Tipo.DESPESA),
```

Executamos com "Alt + Shift + F10".

Repara que temos um aspecto visual diferente da nossa App base, ou seja, o sinal de subtração está na frente do **R$** sendo que na App base fica na frente do número.

Modificaremos novamente a classe que formata para brasileiro, fazendo com que o **negativo** fique depois de `R$` .

Hora de modificar a formatação da moeda para o negativo

Dentro do pacote "extension", selecionamos o arquivo `BigDecimalExtension.kt` para a formatação da moeda.

```
fun BigDecimal.formataParaBrasileiro() : String {
    val formatoBrasileiro = DecimalFormat
        .getCurrencyInstance(Locale(language: "pt", country: "br"))
    return formatoBrasileiro
        .format(obj:this)
        .replace(oldValue: "R$", newValue: "R$ ")
}
```

A única parte que ainda escrevemos visualmente é quando temos apenas o *R$*, e nos queremos colocar um espaço entre o tipo da moeda e a informação.

E então, encadearemos mais um `replace()`, que fará a alteração e irá considerar o seguinte padrão:

- `"-R$ "` para `"R$ -"`

Assim que pegarmos o padrão `"-R$ "`, ele automaticamente será trocado por `"R$ -"`.

```
fun BigDecimal.formataParaBrasileiro() : String {
    val formatoBrasileiro = DecimalFormat
        .getCurrencyInstance(Locale(language: "pt", country: "br"))
    return formatoBrasileiro
        .format(obj:this)
        .replace(oldValue: "R$", newValue: "R$ ")
        .replace(oldValue: "-R$ ", newValue: "R$ -")
}
```

Ao executarmos a *app*, vemos que realmente conseguimos o resultado esperado.

Conseguimos concluir essa primeira tarefa. A segunda tarefa será **lidar com as cores**. Sabemos que as cores da Receita e da Despesa são cores fixas, independente dos valores.

Voltando ao `ResumoView()`, precisamos colocar **cor** no mesmo momento em que adicionamos uma receita ou uma despesa. Como foi visto anteriormente em `ListaTransacoesAdapter()`, para colocar a cor, é preciso realizar uma chamada que pega o `ContextCompat.getColor()`.

Copiamos `ContextCompat.getColor(context, R.color.receita)`, e voltamos para o `ResumoView()` e colamos dentro da função `adicionaReceita()`.

```
    fun adicionaReceita() {
        var totalReceita = resumo().receita()
        ContextCompat.getColor(context, R.color.receita)
        view.resumo_card_receita.text = totalReceita.formataParaBrasileiro()
    }
```

Para usar essa função, é necessário passar uma referência de `context` por parâmetro do construtor primário.

```
class ResumoView(private val context: Context,
                 private val view: View,
                 transacoes: List<Transaco>)
```

Com isso, agora somos capazes de pegar a cor baseando-se no contexto. Na função `adicionaReceita()`, já temos a cor da Receita! Então, agora basta chamarmos o *componente* `view.resumo_card_receita` e usar o `setTextColor()`.

```
    fun adicionaReceita() {
        var totalReceita = resumo().receita()
        view.resumo_card_receita.setTextColor(ContextCompat.getColor(context, R.color.receita))
        view.resumo_card_receita.text = totalReceita.formataParaBrasileiro()
    }
```

Será que irá funcionar? Veremos com o "Alt + Shift + F10".

Olhe só, deu alguns erros no console. Isso aconteceu pois, a partir do momento que colocamos o *context* no contrutor primário, todas as referências que usavam o *ResumoView*, agora terão que mandar o *context*.

Já que estamos na *activity*, mandaremos a própria referência dela.

```
private fun configuraResumo(transacoes: List<Transacao>) {
    val view: View = window.decorView
    val resumoView = ResumoView(context: this, view, transacoes)
    resumoView.adicionaReceita()
    resumoView.adicionaDespesa()
    resumoView.adicionaTotal()
}
```

Iremos testar a aplicação.

Como podemos ver, conseguimos mostrar o valor da **receita**, faremos o mesmo para a **despesa**.

Em `ResumoView()`, podemos usar a mesma estratégia que utilizamos anteriormente, copiando a linha de código que está *setando* a cor da receita.

```
    fun adicionaDespesa() {
        var totalDespesa = resumo().despesa()
        view.resumo_card_despesa.setTextColor(ContextCompat.getColor(context, R.color.despesa))
        view.resumo_card_despesa.text = totalDespesa.formataParaBrasileiro()
    }
```

Vamos executar novamente. Obtemos esse resultado:

Legal, a *app* agora nos mostra os valores da receita e da despesa com cores diferentes. O que está faltando é **atribuir uma cor ao *total***.