Melhorias na Lista de Compras

Android com Kotlin

Flávio Augusto de Freitas

Melhorias na Lista de Compras

Neste projeto vamos implementar algumas alterações no projeto Lista de Compras e fazer a lista de compras melhorada! Faremos algumas atualizações no layout do App, que ficou muito simplificado, depois daremos um upgrade na lista. Em vez de simplesmente inserirmos o nome do produto vamos inserir também a quantidade, o valor do produto e uma foto!

Além disso, incluiremos um texto para exibir o total da compra. A imagem a seguir mostra o resultado final do aplicativo para você ver como ficará o resultado ao final deste projeto:

Figura 1: Lista de compras, aspecto final.

Funcionalidades do app

Vamos começar como sempre pelo planejamento. Nós já temos um aplicativo em funcionamento então o planejamento é somente das alterações. A primeira alteração que podemos fazer é modificar a lista para receber não só a informação de nome do produto, mas também a informação da quantidade, valor e foto.

Pensando desta maneira conseguimos enxergar que precisaremos de mais campos para o usuário inserir todas essas informações. Como agora o usuário precisará cadastrar mais coisas, talvez deixar tudo na mesma tela não seja uma boa ideia, pois a tela ficaria muito poluída. Que tal então dividirmos o aplicativo em duas telas?

Na tela principal, aparecerá a lista com todos os itens inseridos, o total da compra e um botão que dará acesso a uma outra tela somente de inserção de informações. Veja como ficaria o resultado final:

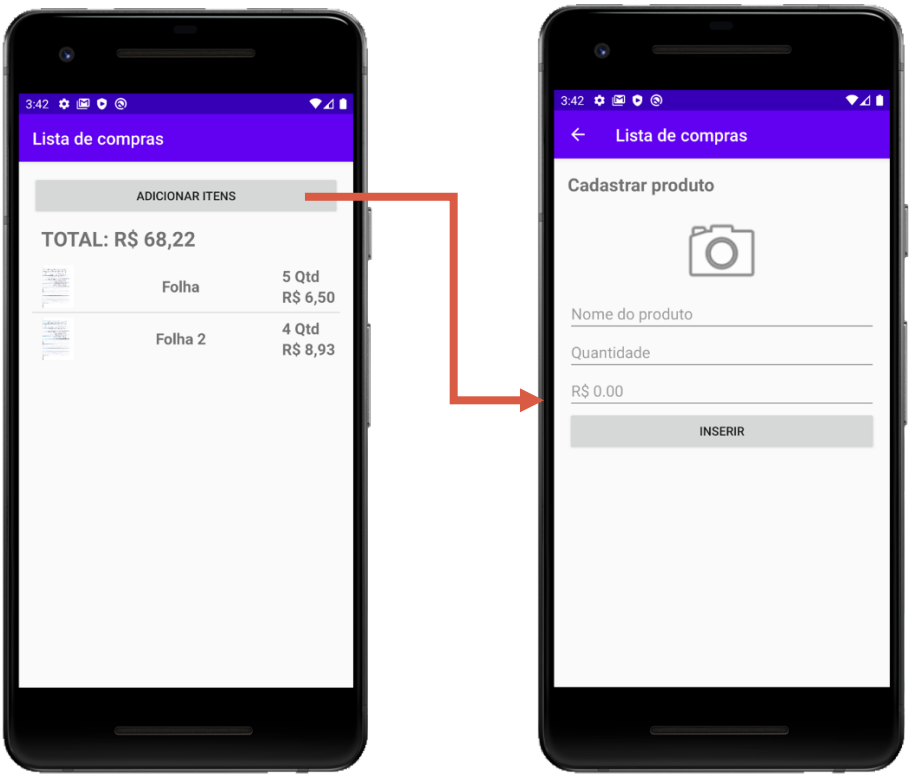

Figura 2: Telas do app.

Desta forma, separamos as responsabilidades: uma tela servirá somente para exibir informações e a outra tela servirá somente para cadastrar informações.

Vamos precisar pensar também no layout da lista. Hoje a lista mostra simplesmente o nome do produto, uma única String, mas agora com novas informações a lista precisará se adequar a um novo layout. Aqui entra uma grande mudança em relação à lista feita no capítulo anterior, lá nossa lista era simples, agora vamos montar uma lista personalizada!

Quando me refiro a listas personalizadas aqui no Android, quero dizer que podemos personalizar o layout da lista com quantas informações desejarmos! Podemos utilizar imagens, tamanhos de fonte diferente nos textos, cores diferentes, planos de fundo etc. Enfim, quando se trata de personalização o Android nos dá total liberdade. Para o nosso caso, vamos personalizar a lista para mostrar a imagem do produto, seu nome, a quantidade e o valor.

Precisamos levar em consideração também que anteriormente quando estávamos guardando somente o nome do produto uma variável do tipo String dava conta, mas e agora, como faremos para agrupar todas essas informações? Para resolver esse problema podemos recorrer à Programação Orientada a Objetos! Que tal criarmos uma classe que tenha todos os atributos de que precisaremos, nome, quantidade, valor e imagem?

Essa solução se mostra muito adequada pois, além de resolver nosso problema no momento, nos dá a liberdade de evoluções futuras. Por ora, temos essas informações, mas e se depois quisermos guardar mais uma informação? Tendo uma classe para isso, bastaria criar um novo atributo!

Vamos ao trabalho!

Criando uma nova Activity de cadastro

Vamos começar separando as telas. Da forma como está agora,o aplicativo faz a inserção e a visualização das informações tudo na mesma tela. Vamos mudar isso. O primeiro a se fazer é criar uma nova Activity, que será a tela de inserção de dados. Deixaremos a MainActivity para mostrar a lista, assim, quando o usuário abrir o aplicativo, ele já vê direto a lista, e se ele quiser cadastrar um novo item ele poderá acessar a tela de cadastro.

Clique com o botão direito em cima da pasta java e escolha a opção New > Activity > Empty Activity. Nomeie o arquivo como CadastroActivity e clique em Finish.

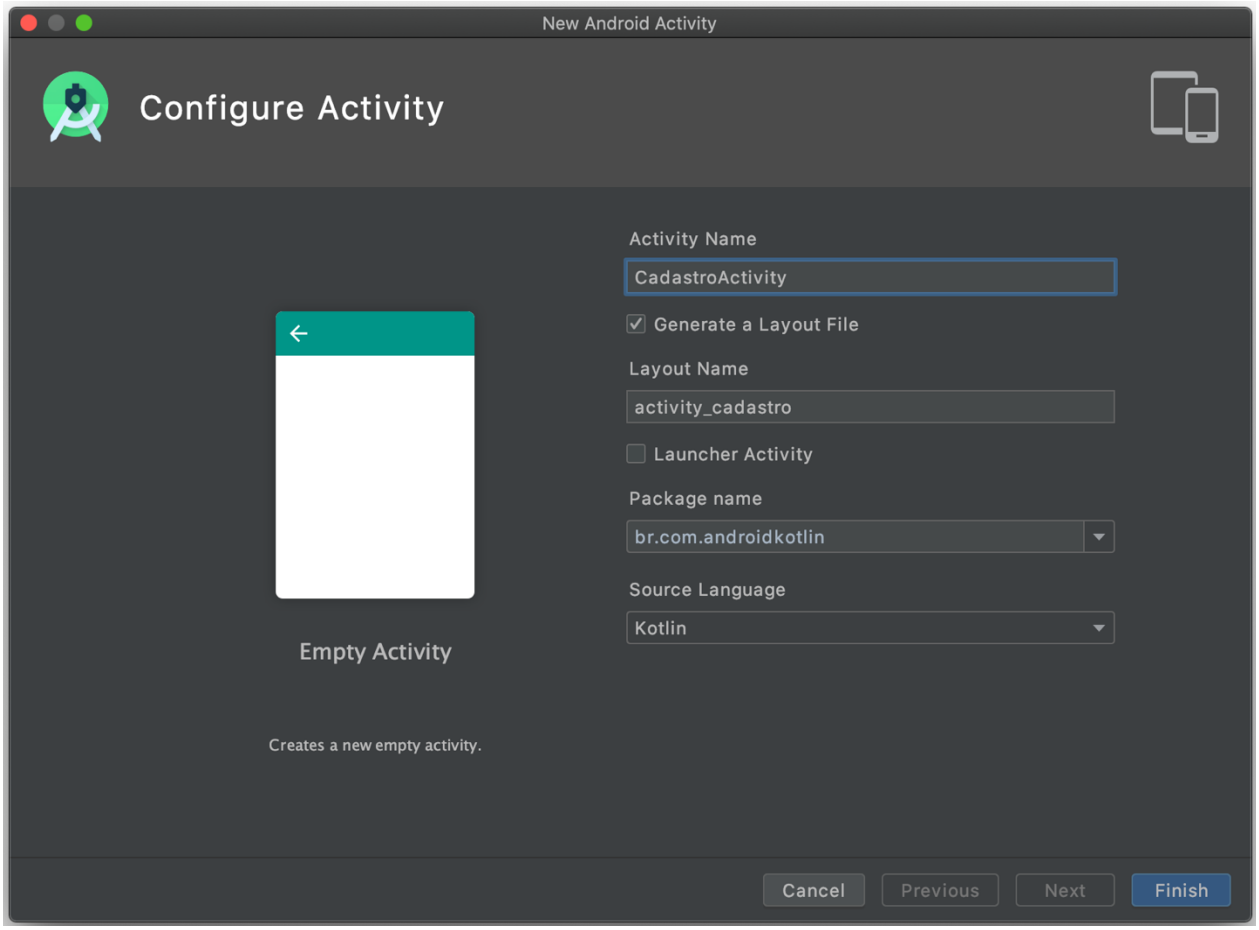

Figura 3: Criando activity Cadastro

Isso vai criar uma nova Activity. Lembre-se de que, fazendo dessa forma, a IDE criará o arquivo Kotlin da ActivityCadastroActivity, o arquivo xml activity_cadastro e também registrará essa Activity no arquivo AndroidManifest.

Agora vamos montar o layout dessa tela. O layout final deve ficar parecido com a imagem a seguir:

Figura 4: Layout Cadastro

Podemos identificar pela imagem que esse layout pode ser feito com um layout linear e orientação vertical, pois os componentes estão um embaixo do outro. Podemos identificar também que precisaremos de um ImageView para exibir a foto, três EditText (para nome, quantidade e valor) e um botão no final para efetuar o cadastro. Temos também um TextView com o texto "Cadastrar produto" para informar o usuário do intuito dessa tela.

Abra o arquivo activity_cadastro para montarmos esse layout. A montagem não tem nada de novo, vamos simplesmente definir todos os componentes em um LinearLayout com orientação vertical.

Seguem as configurações de cada componente do layout:

- LinearLayout:
  - android:orientation="vertical"
  - android:padding="16dp"
- TextView:
  - android:text="Cadastrar produto"
  - android:textStyle="bold" (Para deixar o texto em negrito)
  - android:textSize="21sp"
  - android:layout_marginBottom="15dp"
- ImageView:
  - android:layout_width="100dp"
  - android:layout_height="100dp"
  - android:src="@android:drawable/ic_menu_camera" (ícone de câmera do Android)
  - android:id="@+id/img_foto_produto"
  - android:layout_gravity="center_horizontal"
- EditText:
  - android:layout_width="match_parent"
  - android:layout_height="wrap_content"
  - android:id="@+id/txt_produto"
  - android:hint="Nome do produto"

- EditText:
  - android:layout_width="match_parent"
  - android:layout_height="wrap_content"
  - android:id="@+id/txt_qtd"
  - android:inputType="number" (Para exibir oteclado numérico)
  - android:hint="Quantidade"
- EditText:
  - android:layout_width="match_parent"
  - android:layout_height="wrap_content"
  - android:id="@+id/txt_valor"
  - android:inputType="numberDecimal" (Para exibir o teclado numérico com opção de decimais)
  - android:hint="R$ 0.00"
- Button:
  - android:layout_width="match_parent"
  - android:layout_height="wrap_content"
  - android:text="inserir"
  - android:id="@+id/btn_inserir"

A propriedade inputType, que foi usada nos EditText,define o tipo de entrada da caixa de texto. Na prática, o Android exibirá um tipo de teclado específico de acordo com o inputType escolhido. Por exemplo, se você tem um EditText em que o usuário vai digitar um endereço de e-mail, você deve especificar um inputType igual a textEmailAddress, assim o teclado quese abrirá para o usuário será um teclado que facilita a entrada de e-mails.

Existem diversos tipos de inputType. Os mais comuns e mais úteis no dia a dia são:

- number: Números inteiros
- numberDecimal: Números decimais
- numberSigned: Números com sinal
- textPassword: Senha em texto
- numberPassword: Senha numérica
- phone: Número de telefone
- date: Datas

> *Para uma lista completa sobre os inputTypes consulte a documentação oficial no link: https://developer.android.com/reference/android/text/InputType.html.*

Vamos começar a implementar o código dessa tela. Começaremos pelo container principal, que será um layout linear, e depois vamos adicionando os componentes dentro dele:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context="br.com.livrokotlin.listadecompras.CadastroActivity"
  android:orientation="vertical" android:padding="16dp">
</LinearLayout>
```

Agora vamos adicionando os componentes. Primeiro, vamos colocar um TextView com um título Cadastrar Produto para indicar ao usuário que esse é um formulário de cadastro de produto:

```
<TextView
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:text="Cadastrar produto"
  android:textStyle="bold"
  android:textSize="21sp"
  android:layout_marginBottom="15dp"
  />
```

Em seguida, vamos utilizar o componente ImageView para exibir uma imagem padrão, que depois o usuário usará para personalizar a imagem do produto.

O componente ImageView

O componente ImageView tem como objetivo exibir uma imagem ao usuário, podendo ser um Bitmap ou um arquivo de imagem da pasta drawable. Na pasta res do projeto, há uma pasta chamada drawable, que é onde colocamos imagens para usar no nosso App. Então, para exibir uma imagem no ImageView primeiro devemos ter essa imagem na pasta drawable. Você pode fazer isso simplesmente copiando a imagem e colando na pasta pelo próprio Android Studio. Em seguida, você define a propriedade src do ImageView referenciando essa imagem, veja um exemplo:

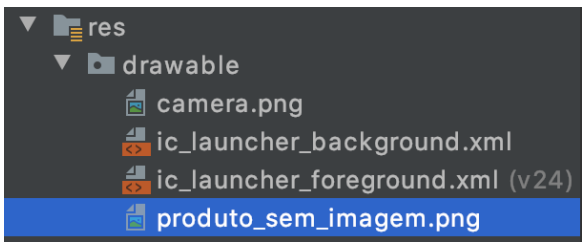

Figura 5: Pasta drawable.

Temos 4 arquivos dentro da pasta, 2 PNGs e 2 XMLs. Os 4 são arquivos de imagens, mas vamos nos focar nos de tipo PNG, que são imagens mais comuns de trabalharmos no dia a dia. Vamos supor que eu queira

exibir em um ImageView a imagem produto_sem_imagem.png. Para isso, eu definiria a propriedade src do ImageView da seguinte maneira: android:src="@drawable/produto_sem_imagem". Veja que eu não preciso passar a extensão da imagem, somente o nome dela. A tag completa desse exemplo ficaria assim:

```
<ImageView
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:src="@drawable/produto_sem_imagem"
  />
```

Voltando ao nosso projeto, vamos implementar também um ImageView abaixo do TextView do título. Nesse ImageView, vamos definir a imagem de um ícone de câmera para indicar ao usuário que ele deve carregar a imagem do produto nesse campo. Para isso, você deveria ter a imagem do ícone de câmera em sua pasta drawable, mas a plataforma Android tem alguns ícones "embutidos" que você pode usar simplesmente utilizando@android:drawable/ mais o nome do recurso. O ícone que vamos utilizar se chama ic_menu_camera. Vamos montar um ImageView que exiba esse ícone:

```
<ImageView
  android:layout_width="100dp"
  android:layout_height="100dp"
  android:src="@android:drawable/ic_menu_camera"
  />
```

Deixei um tamanho fixo de 100dp por 100dp pois acredito ser um tamanho suficiente para esse ícone, mas você pode testar outros valores e ver como fica a visualização. Vou adicionar um id e também deixar essa imagem centralizada:

```
<ImageView
  android:layout_width="100dp"
  android:layout_height="100dp"
  android:src="@android:drawable/ic_menu_camera"
  android:id="@+id/img_foto_produto"
  android:layout_gravity="center_horizontal"

  />
```

Agora vamos colocar os EditText s de entrada de dados, vamos colocar um para o nome do produto, um para a quantidadee um para o valor:

```
<EditText
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:id="@+id/txt_produto"
  android:hint="Nome do produto"
  />

<EditText
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:id="@+id/txt_qtd"
  android:inputType="number"
  android:hint="Quantidade"
  />

<EditText
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:id="@+id/txt_valor"
  android:inputType="numberDecimal"
  android:hint="R$ 0.00"
  />
```

Por fim, vamos incluir um botão para salvar os dados:

```
<Button
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:text="inserir"
  android:id="@+id/btn_inserir"
  />
```

Confira agora como ficará o código completo:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context="br.com.androidkotlin.CadastroActivity"
  android:orientation="vertical"
```

```
  android:padding="16dp">

<TextView
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:text="Cadastrar produto"
  android:textStyle="bold"
  android:textSize="21sp"
  android:layout_marginBottom="15dp"
  />

<ImageView
  android:layout_width="100dp"
  android:layout_height="100dp"
  android:src="@android:drawable/ic_menu_camera"
  android:id="@+id/img_foto_produto"
  android:layout_gravity="center_horizontal" />

<EditText
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:id="@+id/txt_produto"
  android:hint="Nome do produto"
  />

<EditText
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:id="@+id/txt_qtd"
  android:inputType="number"
  android:hint="Quantidade"
  />

<EditText
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:id="@+id/txt_valor"
  android:inputType="numberDecimal"
  android:hint="R$ 0.00"
  />
```

```
<Button
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:text="inserir"
  android:id="@+id/btn_inserir"
  />

</LinearLayout>
```

Agora vamos já criar o esquema de navegação entre as telas, e para isso precisamos definir a hierarquia de telas. Sabemos que a tela principal será a MainActivity e a CadastroActivity será chamada a partir da main, então podemos dizer que a CadastroActivity é uma tela filha da MainActivity. Isso é importante para podermos configurar a hierarquia de telas corretamente, pois o Android se baseará nessas informações para criar o esquema de navegação.

Para definirmos que a CadastroActivity será filha da MainActivity, temos que incluir uma meta tag no AndroidManifest. Essa meta tag é simplesmente uma configuração do aplicativo que é definida por um name e um value. Aqui vamos definir uma meta tag com nameandroid.support.PARENT_ACTIVITY, o que vai dizer ao sistema operacional que essa Activity é filha de outra Activity. Devemos dizer de quem ela é filha através da propriedade value, nesse caso ela será filha da MainActivity.

A definição dessa meta tag ficará assim:

```
<meta-data
  android:name="android.support.PARENT_ACTIVITY"
  android:value=".MainActivity"
  />
```

Inclua essa tag dentro da tag da Activity de cadastro no arquivo AndroidManifest. A tag da activity ficará assim:

```
<activity android:name=".CadastroActivity">
  <meta-data
    android:name="android.support.PARENT_ACTIVITY"
    android:value=".MainActivity"
    />
</activity>
```

Com essa configuração, o Android saberá qual o nosso esquema de navegação e então não precisaremos reinventar a roda, ele próprio já criará um botão de voltar na tela de cadastro que volta para a tela principal!

Agora precisaremos programar como essa tela de cadastro será aberta. No nosso layout inicial, pensamos em incluir um botão na tela principal e, ao clicar nele, a tela de cadastro se abrirá. Vamos manter essa ideia e programar esse botão.

Abra o arquivo activity_main para definirmos o botão no xml e aproveitar para ajeitar o layout dessa tela.

Primeiro, vamos fazer uma limpeza nesse arquivo, alguns itens não precisam mais estar aqui porque estão agora na tela de cadastro. Essa tela terá agora somente um botão, um texto para mostrar o total e a lista. Então vamos remover daqui os componentes EditText e Button, pois não são mais necessários nessa Activity. Deixe somente o LinearLayout e a ListView, seu arquivo ficará assim:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context="br.com.androidkotlin.MainActivity
  android:orientation="vertical"
  android:padding="16dp">

<ListView
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:id="@+id/list_view_produtos">
  </ListView>

</LinearLayout>
```

Agora, adicione uma TextView em cima da lista. Essa TextView exibirá uma somatória com o total dos itens inseridos na lista.

Veja o código:

```
<TextView
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="TOTAL: 0,00"
  android:textSize="24sp"
  android:textStyle="bold"
  android:padding="10dp"
  android:id="@+id/txt_total"
  />
```

Em cima dessa TextView, adicione um botão com texto **Adicionar itens**, que será usado para direcionar o usuário à tela de cadastro. Veja o código do botão:

```
<Button
```

```
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:text="Adicionar itens"
  android:id="@+id/btn_adicionar"
  />
```

Certo, agora já temos o layout dessa tela ajustado. No entanto, com essas modificações de layout o nosso código parou decompilar porque alguns componentes não existem mais na MainActivity, já que foram migrados para a CadastroActivity. Vamos então ajustar o código.

Abra o arquivo MainActivity e localize a seguinte linha de código:

```
btn_inserir.setOnClickListener{
 ...
```

Essa é a definição do listener do botão inserir, mas não faremos mais essa ação aqui, isso tudo ficará na tela de cadastro. Vamos transferir esse pedaço de código para lá.

Selecione todo o bloco desse listener e recorte (Ctrl + X). Com esse pedaço de código recortado, abra o arquivo CadastroActivity e cole-o no método onCreate.

Você vai perceber que a linha produtosAdapter.add(produto) começará a apresentar um erro, isso porque o adaptador não está nessa tela! Por ora, vamos apagar essa linha, faremos outra estratégia para enviar a informação cadastrada ao adaptador.

O código da CadastroActivity ficará assim:

```
import android.os.Bundle
import android.support.v7.app.AppCompatActivity
import kotlinx.android.synthetic.main.activity_cadastro.*

class CadastroActivity: AppCompatActivity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_cadastro)

    //definição do ouvinte do botão
    btn_inserir.setOnClickListener {

      //pegando o valor digitado pelo usuário
      val produto = txt_produto.text.toString()

      if (produto.isNotEmpty()) {
```

```
        //envia o item para a lista


        //limpando a caixa de texto
        txt_produto.text.clear()
      } else {
        txt_produto.error = "Preencha um valor"
      }
    }
  }
}
```

> *Atenção ao:*
>
> `import kotlinx.android.synthetic.main.activity_cadastro.*`
>
> *Como estamos fazendo um Ctrl + X, é possível que a IDE faça esse import automaticamente referenciando a activity_main. Se for o caso, mude para activity_cadastro.*

Agora volte à MainActivity. Vamos implementar o código do botão que abrirá a tela de cadastro. Para isso, dentro do método onCreate adicione um listener para o botão btn_adicionar. Dentro dele, precisaremos executar um comando para abrir a tela de cadastro.

Iniciando uma nova atividade (startActivity)

No Android podemos usar a função startActivity (inicia uma atividade) para abrir uma nova tela do nosso App. Tradicionalmente, o método startActivity recebe por parâmetro uma Intent que pode ser explícita ou implícita. Uma Intent implícita significa que deixaremos o próprio Android decidir qual Activity ele vai abrir. Por exemplo, para abrir uma URL no browser poderíamos fazer o seguinte código:

```
//criando uma Intent implícita para abrir o google.com
val intent = Intent(Intent.ACTION_VIEW, Uri.parse("http://google.com"))


//iniciando a atividade
startActivity(intent)
```

Nesse código, não estamos especificando qual a Activity que vai abrir a URL. O próprio Android vai resolver qual Activity abrir. Como nesse caso é uma URL, a Activity que se abrirá será a do browser.

Quando criamos uma Intent explícita, dizemos exatamente qual activity queremos abrir, nesse caso o objeto Intent deve receber dois parâmetros na sua criação: Um objeto Context e a class da activity que desejamos abrir. No nosso App gostaríamos de abrir explicitamente a tela CadastroActivity, então podemos criar uma Intent assim:

```
//Criando a Intent explícita
val intent = Intent(this, CadastroActivity::class.java)

//iniciando a atividade
startActivity(intent)
```

Essa sintaxe CadastroActivity::class.java é um pouco estranha, não? Realmente, o que acontece aqui é que o objeto Intent recebe um tipo Class que é um tipo do Java e por isso temos que passar desta forma. Vamos ver mais sobre Intents ainda neste projeto. Vamos fazer esse código dentro do listener do btn_adicionar:

```
btn_adicionar.setOnClickListener {
  //Criando a Intent explícita
  val intent = Intent(this, CadastroActivity::class.java)

  //iniciando a atividade
  startActivity(intent)
}
```

Agora você pode testar o aplicativo para ver o que fizemos até aqui. A navegação entre as telas deve estar funcionando, e o layout de ambas as telas também já deve estar pronto.

Personalizando a ListView

Vamos agora fazer a personalização da lista. Diferente da outra lista que só exibia um texto, agora teremos um layout bem mais sofisticado, mas para isso funcionar precisaremos fazer várias modificações no nosso código. Vamos fazer por partes.

O primeiro a se fazer é montar o layout dos itens. Não fizemos isso ainda porque na primeira versão deste projeto nós usamos um layout padrão para uma exibição de texto, que o próprio Android disponibiliza. Mas agora não, como nossa lista é muito particular, precisaremos montar o layout dos itens. Para isso, precisaremos deum arquivo xml que terá a definição do layout e posteriormente ousaremos no adaptador.

Pense nesse arquivo que criaremos como um molde que todos os itens seguirão. O layout que precisamos montar é para obter o seguinte resultado:

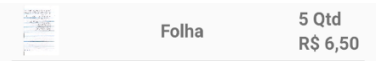

Figura 6: Item da lista.

Perceba que nesse layout tem uma imagem à esquerda, um texto centralizado, e à direita dois textos, um embaixo do outro. Esse é o layout a que precisamos chegar e, uma vez construído, todos os itens da lista seguirão o mesmo padrão.

Então vamos criar um arquivo xml para montar esse layout. Clique com o botão direito na pasta layout e escolha a opção New > Layout Resource file. Na janela que se abrir preencha o campo File name (nome do arquivo) com list_view_item e o campo Root Element (elemento raiz) como LinearLayout e clique em OK.

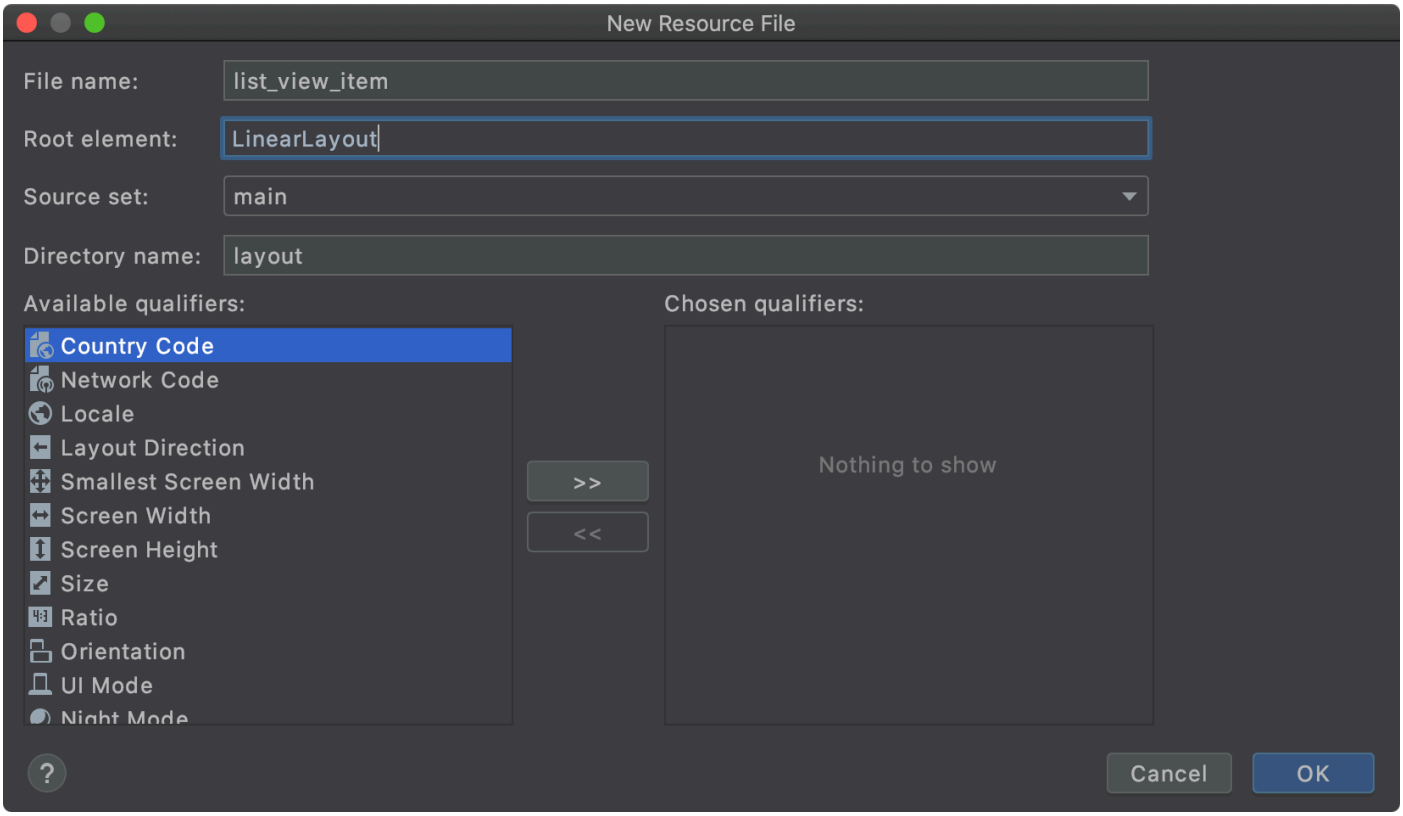

Figura 7: Criando a list_view_item.xml

Isso criará um novo arquivo na pasta, no qual vamos implementar o código do layout dos itens. Para montá-lo, vamos usar um LinearLayout horizontal onde colocaremos os elementos de ImageView para imagem, TextView para o nome do produto e outro LinearLayout, mas agora vertical, que abrigará os outros dois TextView. Observe a imagem a seguir, ela mostra o esqueleto do layout que precisaremos montar:

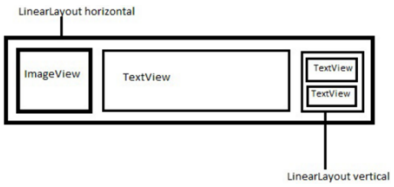

Figura 8: Esquema de item da list_view.

Para a montagem desse layout só existe um ponto que ainda não abordamos, o peso ponderado do layout! O LinearLayout nos permite atribuir um peso para cada componente dentro dele através do atributo layout_weight. Esse atributo define um valor de importância para cada componente na questão de quanto espaço ele deve ocupar na tela. Vamos a um exemplo.

Imagine que você tenha um LinearLayout horizontal com dois componentes dentro. Se você quiser que cada componente ocupe exatamente a metade da tela, basta atribuir peso 1 para os componentes, desta forma o LinearLayout fará uma ponderação nos pesos e, como ambos terão pesos iguais, dividirá o espaço igualmente para cada componente.

Para esse atributo funcionar corretamente, devemos atribuir a largura ou altura do componente para 0dp, então se eu quero balancear componentes na horizontal eu devo atribuir o peso deles e deixar a largura igual a0dp (layout_width="0dp"). No entanto, se eu estou balanceando um componente na vertical, eu devo atribuir o seu peso e deixar a sua altura como 0dp

(layout_height="0dp"). A imagem a seguir representa um linear layout com 2 componentes dentro balanceados com peso 1:

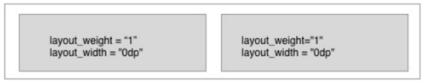

Figura 9: Layout balanceado com peso 1.

Para montar o layout, vamos utilizar um LinearLayout com orientação horizontal, dentro do qual criaremos uma ImageView com tamanho fixo de 50x50. Em seguida, criaremos um TextView com peso 1, assim ele dominará o espaço restante. Por fim, criaremos um LinearLayout com orientação vertical e dentro dele 2 elementos TextView.

O código a seguir demostra como criar esse layout. Abra o arquivo list_view_item e construa esse código:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="horizontal"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:minHeight="50dp"
  android:gravity="center_vertical"
  android:padding="5dp">

  <ImageView
    android:layout_width="50dp"
    android:layout_height="50dp"
    android:src="@android:drawable/ic_menu_camera"
    android:id="@+id/img_item_foto"
    />

  <TextView
    android:layout_width="0dp"
    android:layout_weight="1"
    android:layout_height="wrap_content"
    android:id="@+id/txt_item_produto"
    android:textSize="18sp"
    android:textStyle="bold"
    android:gravity="center"
    />

  <LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <LinearLayout
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:orientation="horizontal">
```

```
      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/txt_item_qtd"
        android:textSize="18sp"
        android:textStyle="bold"
        />

      <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="18sp"
        android:text=" Qtd"
        android:textStyle="bold"
        />
    </LinearLayout>

    <TextView
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:textStyle="bold"
      android:textSize="18sp"
      android:id="@+id/txt_item_valor"
      />

  </LinearLayout>
</LinearLayout>
```

Perfeito! Já temos o layout dos itens da nossa lista! Vamos seguir adiante para fazer a lista funcionar e ver se o resultado visual está bom ou precisará de algum ajuste - e caso precise voltaremos nesse arquivo para os ajustes visuais.

Criando a classe de dados

O próximo passo agora é definir a estrutura de dados que esta lista vai representar, isto é, a estrutura das informações que ela vai exibir! Anteriormente, nossa estrutura de dados era um texto simples, então usamos o objeto String como estrutura de dados. Mas e agora? Com uma simples String conseguiremos representar todas as novas informações?

Nesse caso, uma String já não nos atenderá mais, então precisaremos de uma estrutura para essa demanda específica. Como o Kotlin é uma linguagem orientada a objetos, podemos criar uma classe que representará exatamente a estrutura de dados de que precisamos!

Vamos primeiro pensar no nome dessa classe, que deve ser claro o bastante para indicar exatamente o que essa classe representa. O nome Produto me parece um nome adequado, uma vez que essa classe representará um produto inserido na lista. Vamos pensar também em quais atributos essa classe terá.

Certamente, ela terá um atributo para guardar o nome do produto, que podemos chamar de nome, e ele deve ser do tipo String pois o nome do produto é um conteúdo de texto. Precisaremos também de um atributo para

a quantidade, que podemos chamar de quantidade, esse será do tipo Int pois a quantidade será um número inteiro.

Vamos precisar também de um atributo para o valor do produto, que podemos chamar de valor e será do tipo Double pois o valor de um produto não é um número inteiro (muitas vezes, um preço é um número decimal) então precisaremos de uma variável que suporte esse tipo de número. Por fim, vamos definir um atributo para guardar a foto do produto e chamaremos de foto. Esse atributo pode ser do tipo Bitmap, tipo de variável que serve para guardar elementos gráficos, como uma foto.

Para criação da classe, clique com o botão direito em cima da pasta br.com.androidkotlin e escolha a opção New > Kotlin File/class.

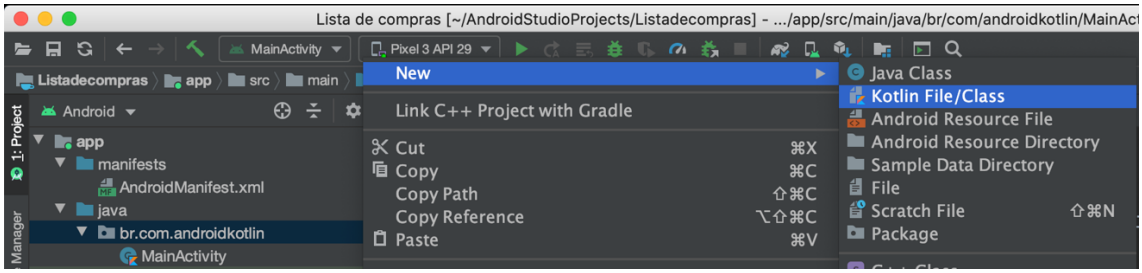

Figura 10: Criando uma classe usando menu de contexto.

Na janela que se abrir, preencha o campo Name (nome) como Produto e clique em OK.

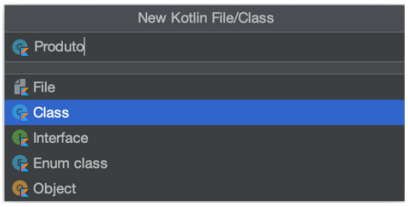

Figura 11: Classe Produto (no caso do macOS, basta digitar o nome da classe, escolher Class na lista e pressionar ENTER).

Isso criará um arquivo vazio na pasta. Abra-o para começar a implementação do código. Podemos dizer que a classe que vamos criar é uma classe de dados, isto é, que usaremos somente para guardar informações; não será uma classe que terá métodos ou funções, somente os atributos nome, quantidade, valor e foto. Por isso, podemos criá-la como uma data class e assim nosso código ficará muito reduzido!

Para criar a nossa classe, iniciaremos com as palavras data class, indicando que é uma classe de dados. Em seguida, digitaremos o nome da classe, Produto, depois implementaremos através do construtor os atributos de nome, quantidade,valor e foto. O código completo da classe será o seguinte:

```
data class Produto(val nome:String, val quantidade:Int, val valor:Double, val foto: Bitmap )
```

Faça também o import da classe Bitmap:

```
import android.graphics.Bitmap
```

A classe está pronta! Mas vamos pensar em uma situação: será que devemos obrigar o usuário a definir sempre uma foto para o produto? Acredito que esse atributo deva ser opcional, assim, se o usuário não definir nenhuma foto, usaremos alguma imagem padrão.

Pensando nisso, vamos mudar um pouco essa classe e deixar o atributo foto aceitando valores nulos, pois se nenhuma foto for atribuída essa variável deve ser nula, e também vamos deixar esse atributo como opcional na construção do objeto.

Para deixar o atributo foto aceitar valores nulos, basta incluir o sinal de interrogação na frente de sua definição: val foto: Bitmap?. Agora vamos deixá-lo como atributo opcional; vamos definir que se o usuário não passar nenhum valor a essa variável ele assumirá o valor nulo, por padrão. Para isso, basta adicionar = null na definição da variável, assim: val foto: Bitmap? = null. Feito isso nossa classe agora ficará assim:

```
data class Produto(val nome:String, val quantidade:Int, val valor:Double, val foto: Bitmap? = null)
```

Customizando o adaptador

Agora faremos talvez o passo mais importante, que é a customização do adaptador! Com todas essas alterações, o adaptador que estamos usando já não dá mais conta. Lembrando da analogia de que um adaptador é como um adaptador de tomada, desses que usamos para ligar uma tomada de 3 pinos em uma rede elétrica de 2 pinos, agora estaríamos lidando com uma situação totalmente nova, uma situação em que nenhum adaptador desses comprados em mercado serviria - imagine que agora nossa rede elétrica é de 4 pinos!

E você deve estar pensando "Pera lá, não existem redes de 4 pinos...". Sim, aí é que está, essa é a liberdade que temos quando trabalhamos com listas e adaptadores no Android. Podemos criar qualquer estrutura de dados que atenda a nossa necessidade e depois simplesmente criar um adaptador customizado para essa estrutura de dados! Essa é a situação que temos agora: imagine que a nossa classe Produto seja a rede elétrica de 4 pinos e, como não existem adaptadores para 4 pinos, vamos criar um adaptador customizado!

Vamos começar criando o arquivo para ele, para isso clique com o botão direito em cima da pasta br.com.androidkotlin e escolha a opção New > Kotlin File/Class. Preencha o nome do arquivo como ProdutoAdapter, pois este será um adaptador exclusivo para a classe Produto.

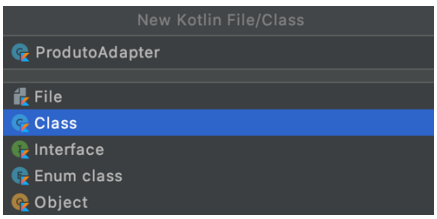

Figura 12: Classe ProdutoAdapter (no caso do macOS, basta digitar o nome da classe, escolher Class na lista e pressionar ENTER).

No arquivo que se abriu, vamos começar a escrever nosso código, vamos iniciar criando a classe: class ProdutoAdapter(contexto: Context). No construtor da classe eu já estou passando um parâmetro do tipo Context chamado contexto. Essa variável receberá o contexto em que esse adaptador está sendo usado.

Em seguida, faremos uma herança da classe ArrayAdapter, isso porque esse adaptador customizado ainda será um adaptador, então ele terá todas as características de um adaptador do tipo ArrayAdapter, mais as características novas que implementaremos. O código ficará o seguinte:

```
import android.widget.ArrayAdapter
import android.content.Context

class ProdutoAdapter(contexto: Context): ArrayAdapter<Produto>()
{
}
```

No entanto, se você resgatar o código em que criamos o adaptador de String, você verá que a classe ArrayAdapter recebe 2 parâmetros, um contexto e o layout. Veja o código que utilizamos na MainActivity para relembrar:

```
val adapter = ArrayAdapter<String>(this, android.R.layout.simple_list_item_1)
```

> *A palavra-chave this faz referência à instância do objeto em que está sendo utilizada. O construtor do ArrayAdapter recebe como primeiro parâmetro um objeto Context, que é base para o objeto Activity, então ao utilizar o this dentro de uma Activity estamos fazendo uma referência à própria Activity, mas também a um objeto Context.*

Nesse código estamos passando o contexto através da variável this e o layout através doandroid.R.layout.simple_list_item_1. Isso significa que um ArrayAdapter precisa de pelo menos essas duas informações, então, na declaração da nossa classe, precisaremos passá-las. Para a variável de contexto está fácil pois já definimos uma variável contexto, mas o layout será customizado, e isso faremos um pouco mais adiante no código. Por isso, por enquanto passaremos0 no construtor para informar que, por enquanto, não tem nenhum layout definido. O código ficará assim:

```
import android.widget.ArrayAdapter
import android.content.Context

class ProdutoAdapter(contexto: Context): ArrayAdapter<Produto>(contexto,0) {

}
```

Agora precisaremos implementar o comportamento desse adaptador, isto é, como ele fará para preencher as informações que queremos da forma correta.

A classe ArrayAdapter implementa o método chamado getView, que é o método principal do adaptador porque a função dele é retornar a View de cada item da lista que será exibida na tela. O método getView é responsável por montar o item de exibição na tela, é esse método que pegará a informação e colocará no lugar certo, por isso ele é o método mais importante de um adaptador.

Para entender melhor como o método getView funciona, imagine uma linha de produção de veículos em que existe um monte de peças separadas, e na linha de montagem é feita a junção de todas as peças para, no final, sair um veículo pronto. É exatamente assim que funciona um adaptador com o método getView. Nesse caso, o getView seria a linha de montagem em que chegam todas as informações e ele será responsável por fazer a junção de tudo e, ao final, devolver um item pronto para exibição.

Ainda neste mesmo exemplo, imagine que esta fábrica vai produzir um total de 10 veículos por dia, dessa forma, na linha deprodução serão montados exatamente 10 veículos, assim acontece também com o método getView: se temos 10 itens para exibir na lista, o getView será executado 10 vezes e montará cada um dos10 itens!

A estrutura do método getView é a seguinte:

```
override fun getView(position: Int, convertView: View?, parent: ViewGroup): View { }
```

Adicione também os imports na classe:

```
import android.view.View
import android.view.ViewGroup
```

> *Dica: o Android Studio possui um recurso chamado AutoImport, que basicamente faz os imports automaticamente no nosso projeto conforme formos usando os componentes. Para habilitá-lo clique no menu File > Settings e, na caixa de pesquisa. Busque por auto import. Na janela de configuração, selecione All na opção Insert imports onpaste e marque as opções: Add unambiguous imports onthe fly e Optimize imports on the fly.*

Pela estrutura do método, podemos perceber que ele recebe 3 variáveis:

- position: A posição do item que está prestes a ser montado na lista.
- convertView: A View que foi usada no último item para ser reutilizada. Antes de utilizar essa variável devemos sempre verificar se ela não está nula, pois, se for o primeiro item a ser criado, essa variável estará nula e não será possível reutilizá-la.
- parent: Uma view pai à qual eventualmente a view principal poderá estar atrelada.

Vamos começar por partes. Primeiro vamos declarar uma variável do tipo View que será responsável por guardar o layout desse item. Vou chamá-la de v para facilitar o código do restante do método.

```
val v:View
```

Agora vamos fazer uma verificação em cima da variável convertView e, caso ela não seja nula, vamos preencher a variável v com a variável convertView ; caso contrário, vamos inflar o layout que criamos. Essa ideia é de reaproveitamento da View: se eu inflar o layout na primeira vez, nas próximas ele já vai estar inflado para eu utilizar.

> *Chamamos "Inflar um layout" o processo de converter um arquivo xml para um objeto do tipo View. Imagine que, para fazer um bolo, você precise da massa de bolo e de uma forma untada.*

> *Para assá-lo, você pega a massa e coloca dentro da forma. Isso seria o "inflar o layout", pegar o xml e colocar dentro de um objeto.*

Vamos então à verificação:

```
if(convertView != null){
  v = convertView
} else {
  //inflar o layout
}
```

Então, caso a convertView não seja nula, não precisaremos inflar o layout pois ele já estará inflado dentro da convertView.Agora no else, devemos inflá-lo. Para isso, existe uma classe no Android chamada LayoutInflater, que serve justamente para esse propósito.

Para usar o LayoutInflater, primeiro devemos criar o objeto a partir do contexto: LayoutInflater.from(context).Em seguida, chamamos o método inflate.

O método inflate possui duas assinaturas: a primeira recebe dois parâmetros, que são respectivamente uma View chamada resource e uma ViewGroup chamada root:

```
inflate(resource:View, root:ViewGroup)
```

E a segunda recebe, além do resource e do ViewGroup, um Boolean chamado attachToRoot, que indicará se o layout passado na variável resource será anexado ao ViewGroup:

```
inflate(resource:View, root:ViewGroup, attachToRoot:Boolean)
```

Sabendo que o método inflate pode receber dois ou três parâmetros, vamos entender o que é exatamente cada um.

O primeiro parâmetro é o resource, que é o arquivo xml que você gostaria de "inflar", para nosso caso, será o arquivo list_view_item pois é ele que definimos como layout dos itens dessa lista.

O segundo parâmetro é um ViewGroup de que esse layout faz parte. Aqui temos a opção de simplesmente passar null a essa variável e indicar que nosso layout não faz parte de nenhum ViewGroup, nesse caso teríamos um código assim:

```
LayoutInflater.from(context).inflate(R.layout.list_view_item, null)
```

> *ViewGroup é um tipo especial de View que pode conter outras Views dentro. Um exemplo é o LinearLayout, dentro do qual podemos agregar várias outras Views. Em resumo, uma ViewGroup é um componente que agrega outros componentes dentro dele.*

Quando utilizamos o método inflate dessa forma, estamos dizendo que nosso layout não faz parte de nenhum ViewGroup, oque nesse caso não é bem uma verdade. Aqui, o ViewGroup do qual esse layout faz

parte é a própria ListView, e apesar de esse código funcionar, se utilizarmos dessa maneira o nosso layout acabará não herdando as configurações gerais definidas na ListView.

Se repararmos, o próprio método getView já nos dá acesso a uma variável chamada parent, que é justamente a ListViewque estamos montando, então podemos passar essa variável como ViewGroup do método inflate:

```
LayoutInflater.from(context).inflate(R.layout.list_view_item, parent)
```

Agora precisamos passar o último parâmetro attachToRoot,um valor booleano indicando se o nosso layout será anexado ao ViewGroup. No nosso caso é false, ou seja, não será diretamente anexado ao ViewGroup, mas, sim, herdará configurações gerais definidas na ListView. O código ficará assim:

```
LayoutInflater.from(context).inflate(R.layout.list_view_item, parent, false)
```

Colocando esse código no nosso projeto, teremos o seguinte:

```
...
else {
  //inflar o layout
  v = LayoutInflater.from(context).inflate(R.layout.list_view_item, parent, false)
}
```

Faça também o import da classe LayoutInflater:

```
import android.view.LayoutInflater
```

> *Imaginando que o método getView seja uma linha de produção de automóveis, inflar o layout seria preparar o chassi do veículo para receber os outros componentes.*

Agora, o próximo passo é acessar os dados que serão exibidos na tela. Essa informação já está no adaptador e pode ser acessada pelo método getItem passando a posição do item que desejamos acessar. Para saber a posição do item não há problema, pois o método getView já nos dá acesso a uma variável chamada position, que guarda exatamente a informação de posição que precisamos. Então o código é muito simples:

```
val item = getItem(position)
```

Agora vamos acessar os elementos visuais. Na montagem do layout, colocamos três TextViews: txt_item_produto,txt_item_qtd e txt_item_valor para exibir o nome do produto, a quantidade e o valor, respectivamente. Também colocamos um ImageView, img_item_foto, para exibir a foto do produto.

Então vamos acessar cada um desses elementos através dos Ids. Para isso, vamos usar o método findViewById que já conhecemos, a única diferença é que agora ele será usado na variável v porque é nela que o layout está inflado. Veja:

```
val txt_produto = v.findViewById<TextView>(R.id.txt_item_produto)
val txt_qtd = v.findViewById<TextView>(R.id.txt_item_qtd)
val txt_valor = v.findViewById<TextView>(R.id.txt_item_valor)
```

```
val img_produto = v.findViewById<ImageView>(R.id.img_item_foto)
```

Imports necessários:

```
import android.widget.ImageView
import android.widget.TextView
```

Neste momento, temos acesso a duas coisas importantes:

1. os componentes da tela
2. a variável item que guarda o produto.

Sendo assim, agora basta atualizarmos a propriedade textdos TextViews com as informações da variável item:

```
txt_qtd.text = item.quantidade.toString()
txt_produto.text = item.nome
txt_valor.text = item.valor.toString()
```

Para atualizar a foto, devemos antes verificar se a variável não é nula, pois o usuário não é obrigado a definir uma imagem. E caso a foto for diferente de null, atribuiremos a ela a img_produtoatravés do método setImageBitmap. Esse método serve para atribuir uma variável do tipo Bitmap a um ImageView, que é exatamente a situação que temos.

```
if (item.foto != null){
  img_produto.setImageBitmap(item.foto)
}
```

Por fim, vamos retornar a variável v. A variável v está carregando dentro de si todo o layout do item que será mostrado na lista, então tudo o que foi construído nesse método está sendo carregado dentro da variável v definida no início do método.

```
return v
```

> *Ainda pensando em uma linha de produção, o método getItem seria preparar o motor do veículo. O findViewById seria preparar a pintura e estética, e a atualização dos atributos seria a finalização, quando os componentes são juntados para criar o produto final.*

Veja o código completo da classe:

```
import android.content.Context
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.ArrayAdapter
import android.widget.ImageView
```

```
import android.widget.TextView

class ProdutoAdapter(contexto: Context): ArrayAdapter<Produto>(contexto,0) {

  override fun getView(position: Int, convertView: View?, parent: ViewGroup): View {

    val v:View

    if(convertView != null) {
      v = convertView
    } else {
      v = LayoutInflater.from(context).inflate(R.layout.list_view_item, parent, false)
    }

    val item = getItem(position)
    val txt_produto = v.findViewById<TextView>(R.id.txt_item_produto)
    val txt_qtd = v.findViewById<TextView>(R.id.txt_item_qtd)
    val txt_valor = v.findViewById<TextView>(R.id.txt_item_valor)
    val img_produto = v.findViewById<ImageView>(R.id.img_item_foto)

    txt_qtd.text = item.quantidade.toString()
    txt_produto.text = item.nome
    txt_valor.text = item.valor.toString()

    if (item.foto != null) {
      img_produto.setImageBitmap(item.foto)
    }

    return v
  }
}
```

Nosso adaptador está pronto! Mas tem mais uma pequena coisa que eu gostaria já de ajustar aqui, porque isso dará diferença no resultado final do nosso aplicativo. Definimos que vamos exibir o valor de cada produto de acordo com o que o usuário cadastrar, no entanto, o usuário vai cadastrar um número decimal, por exemplo: 10.50. Para a exibição, seria muito interessante exibir esse número no padrão moeda brasileira R$ 10,50, então vamos aproveitar que já estamos mexendo no adaptador e fazer esse ajuste também.

Para fazer essa formatação de moeda temos a classe NumberFormat em que podemos definir N tipos de formatações numéricas. Essa classe possui um método chamado getCurrencyInstance, que retorna uma instância de formatação do formato moeda! Então podemos usá-lo para construir um objeto de formatação e depois formatar a variável valor.

```
//obtendo a instância do objeto de formatação
val f = NumberFormat.getCurrencyInstance()
```

```
//formatando a variável no formato moeda
txt_valor.text = f.format(item.valor)
```

Devemos nos atentar para um detalhe: o método getCurrencyInstance vai formatar de acordo com a linguagem em que o aparelho estiver configurado. Geralmente, os emuladores estão em inglês, então esse código formataria o valor em dólar e não em reais. Podemos definir qual linguagem de formatação usar passando um objeto Locale para o método getCurrencyInstance. Veja como definir a formatação em português do Brasil:

```
//obtendo a instância do objeto de formatação
val f = NumberFormat.getCurrencyInstance(Locale("pt", "br"))

//formatando a variável no formato moeda
txt_valor.text = f.format(item.valor)
```

Faça os imports da classe NumberFormat:

```
import java.text.NumberFormat
```

E da classe Locale:

```
import java.util.Locale
```

Dessa forma, independentemente da linguagem do aparelho, a formatação será feita sempre em reais. Agora sim nosso adaptador está pronto!

Cadastrando itens

Vamos agora resolver a tela de cadastro. Já temos todo o layout pronto, só precisaremos trabalhar no código. Mas antes precisamos pensar em um detalhe: onde as informações de produtos serão armazenadas?

No Android, cada Activity é independente, ou seja, por padrão elas não compartilham informações. Portanto, temos que pensarem como vamos armazenar essas informações para mostrar na tela principal. O ideal seria usar uma estrutura de banco de dados pois as informações ficariam gravadas permanentemente no disco e qualquer classe do aplicativo poderia acessá-las, mas vamos aprender a utilizar o banco de dados do Android somente no próximo capítulo, então aqui precisamos de uma solução mais simplificada.

Podemos armazenar as informações de produtos em uma variável global, que pode ser acessada de qualquer parte do aplicativo. Com uma variável global, eu posso gravar informações por meio da tela de cadastro e depois lê-las através da tela principal, o ponto negativo desta abordagem é que, como esses dados ficam somente em memória RAM, quando o aplicativo for fechado esses dados serão perdidos, mas isso resolveremos no próximo capítulo com a implementação de um banco de dados.

Em Kotlin, é muito fácil criar variáveis globais. Basta criar um arquivo simples em Kotlin e definir a variável lá, que ela automaticamente estará visível em todo o projeto! Vamos fazer isso então.

Crie um arquivo chamado Utils no projeto. Para isso, basta clicar com o botão direito na pasta br.com.androidkotlin e escolher a opção New> Kotlin file/Class.

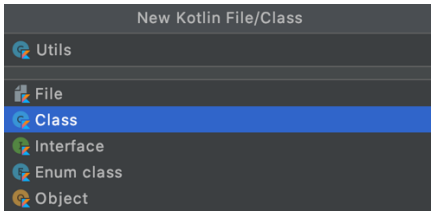

Figura 13: Classe Utils (no caso do macOS, basta digitar o nome da classe, escolher Class na lista e pressionar ENTER).

Vamos chamar esse arquivo de Utils pois podemos usá-lo posteriormente para colocar funções ou outras variáveis que serão úteis ao nosso projeto.

Nesse arquivo, vamos definir a nossa variável global. Essa variável deve ser capaz de armazenar vários produtos cadastrados, então ela não pode ser simplesmente do tipo Produto, ela deve ser uma lista pois não sabemos quantos produtos o usuário deseja cadastrar.

Vamos definir então uma lista mutável, ou seja, uma lista que pode ter sua estrutura alterada incluindo ou removendo itens. Vamos chamar essa variável de produtos Global para dar a ideia de que ela armazenará produtos de forma global e vamos definir que ela será do tipo mutableListOf<Produto>, ou seja, uma lista mutável do tipo Produto. Veja como ficará a implementação:

```
val produtosGlobal = mutableListOf<Produto>()
```

E pronto! Já temos uma variável que pode ser acessada de forma global.

Partiremos agora para o código da tela de cadastro. Abra o arquivo CadastroActivity. Nessa classe, já tem um pedaço de código pronto que fizemos no começo do capítulo, no qual temos a definição do botão e uma verificação para saber se o usuário deixou o nome em branco. O código que temos até agora é o seguinte:

```
//definição do ouvinte do botão
btn_inserir.setOnClickListener {
  //pegando o valor digitado pelo usuário
  val produto = txt_produto.text.toString()

  if (produto.isNotEmpty()) {
    //envia o item para a lista

  } else {
    txt_produto.error = "Preencha um valor"
  }
}
```

Vamos incrementar esse código com as mudanças necessárias. Primeiro, vamos pegar as informações de quantidade e valor – a foto do produto faremos por último. Vamos incluir então uma linha de código para pegar o valor e uma linha para pegar a quantidade:

```
//pegando os valores digitados pelo usuário
val produto = txt_produto.text.toString()
val qtd = txt_qtd.text.toString()
val valor = txt_valor.text.toString()
```

Agora vamos mudar um pouco a verificação também. Antes, verificávamos somente se o nome do produto não estava vazio; agora precisamos verificar as variáveis qtd e valor, pois ele será obrigado a informar todas essas informações. Vamos utilizar operador "E" (&&) para incluir as verificações no mesmo if:

```
if (produto.isNotEmpty() && qtd.isNotEmpty() && valor.isNotEmpty()) {
  //enviado o item para a lista

}
...
```

Dessa forma, só vamos entrar nesse if se todas as variáveis estiverem preenchidas; se por acaso qualquer uma estiver vazia, o código cairá no else. Já aproveitando, vamos ter que mexer nesse else também. Atualmente, o else considera que o nome do produto está vazio e então envia uma mensagem de erro.

Mas agora as coisas são diferentes. O usuário pode ter deixado de preencher qualquer uma das informações, ou pode não ter preenchido nenhuma, são várias situações diferentes que podem acontecer aqui e precisamos saber exatamente qual a caixa que ele deixou de preencher para definir a mensagem correta na caixa correta. Para isso, podemos abrir outros ifs aqui dentro. Que tal se fizermos uma verificação para cada caixinha individualmente e setarmos uma mensagem personalizada? O código ficaria assim:

```
...
} else {
  if (txt_produto.text.isEmpty()) {
    txt_produto.error = "Preencha o nome do produto"
  } else {
    txt_produto.error = null
  }
  if (txt_qtd.text.isEmpty()) {
    txt_qtd.error = "Preencha a quantidade"
  } else {
    txt_qtd.error = null
  }
```

```
  if (txt_valor.text.isEmpty()) {
    txt_valor.error = "Preencha o valor"
  } else {
    txt_valor.error = null
  }
}
```

Nesse código é feita a verificação de cada caixinha. Se ela estiver vazia, colocamos uma mensagem personalizada, caso contrário definimos seu atributo error como null para não aparecer mais erro naquela caixa de texto.

Apesar de fazer sentido, ficou um código um pouco extenso para uma coisa simples. Será que não conseguimos fazer a mesma coisa com menos código? A resposta é sim! Graças à linguagem Kotlin, conseguiremos reduzir esse código todo a 3 linhas!

Isso porque o Kotlin nos permite fazer um if em uma única linha. A estrutura é basicamente a mesma, no entanto, o if e o else são feitos na mesma linha. Esse tipo de if só faz sentido se for uma atribuição de valor condicional, que é nosso caso: aqui estamos atribuindo ao atributo error uma mensagem personalizada ou null, então conseguimos fazer essa atribuição condicional na mesma linha. Veja como fica no código:

```
...
} else {
  txt_produto.error = if (txt_produto.text.isEmpty()) "Preenchao nome do produto" else null
  txt_qtd.error = if (txt_qtd.text.isEmpty()) "Preencha a quantidade" else null
  txt_valor.error = if (txt_valor.text.isEmpty()) "Preencha o valor" else null
}
```

Uma solução bem elegante, não é?

Agora precisamos fazer o cadastro em si, ou seja, pegar essas informações e inserir naquela variável global que criamos. Mas precisamos ter em mente que a variável produtosGlobal que criamos é uma lista do tipo Produto, sendo assim só podemos inserir nela objetos do tipo Produto. Então, antes de mandar para a lista, precisamos criar esse objeto. Como já temos a classe pronta, basta criarmos um objeto com as informações que o usuário inseriu. Vamos criá-lo:

```
val prod = Produto(produto, qtd.toInt(), valor.toDouble())
```

Nesse código estamos criando um objeto do tipo Produto e passando o nome, a quantidade e o valor. Perceba que, na variável qtd, utilizamos o método toInt() porque definimos essa variável como Int. O mesmo acontece com a variável valor, mas aqui utilizamos o método toDouble() porque definimos que esse atributo é Double. Por fim, não passamos a imagem porque ainda não a temos e, como definimos como parâmetro opcional, o código funcionará mesmo sem ela.

Feito isso, basta adicionar esse item à variável produtosGlobal com o método add:

```
produtosGlobal.add(prod)
```

Na sequência, vamos também limpar as caixinhas para que o usuário possa inserir outro produto:

```
txt_produto.text.clear()
txt_qtd.text.clear()
txt_valor.text.clear()
```

O código completo do botão ficará assim:

```
//definição do ouvinte do botão
btn_inserir.setOnClickListener {

  //pegando os valores digitados pelo usuário
  val produto = txt_produto.text.toString()
  val qtd = txt_qtd.text.toString()
  val valor = txt_valor.text.toString()

  //verificando se o usuário digitou algum valor
  if (produto.isNotEmpty() && qtd.isNotEmpty() && valor.isNotEmpty()) {

    val prod = Produto(produto, qtd.toInt(), valor.toDouble())

    produtosGlobal.add(prod)

    txt_produto.text.clear()
    txt_qtd.text.clear()
    txt_valor.text.clear()

  } else {

    txt_produto.error = if (txt_produto.text.isEmpty()) "Preencha o nome do produto" else null
    txt_qtd.error = if (txt_qtd.text.isEmpty()) "Preencha a quantidade" else null
    txt_valor.error = if (txt_valor.text.isEmpty()) "Preencha o valor" else null
  }
}
```

Você pode testar o aplicativo agora, cadastrar itens e testar se as verificações funcionam. Mas ainda não verá nenhum item cadastrado pois ainda não arrumamos a tela principal para exibi-los.

Exibindo itens cadastrados

Exibir os itens cadastrados na tela principal não será difícil porque já temos tudo o que precisamos pronto, é só começar ajuntar as peças desse quebra-cabeça. Abra o arquivo MainActivity para começarmos os trabalhos.

Vamos começar mudando o adaptador. Vamos utilizar nosso adaptador personalizado que criamos no começo deste projeto.

Localize a linha da definição do adaptador onde está o seguintecódigo:

```
//Implementação do adaptador
val produtosAdapter = ArrayAdapter<String>(this, android.R.layout.simple_list_item_1)
```

Vamos mudar essa definição para uma instancia da classe ProdutoAdapter:

```
//Implementação do adaptador
val produtosAdapter = ProdutoAdapter(this)
```

Em seguida, vamos acessar a variável produtos Global e inserir seus itens no produtosAdapter. A única consideração que temos que fazer é que o método add do adapter aceita somente 1elemento, e a variável produtosGlobal é uma lista; para esses casos, temos o método addAll, que recebe uma lista como parâmetro e adiciona todos elementos:

```
produtosAdapter.addAll(produtosGlobal)
```

Só com isso já conseguimos visualizar os itens cadastrados! Teste o aplicativo e veja se eles aparecem na tela principal.

Aqui podem acontecer duas situações: a primeira é você cadastrar um item, clicar no botão voltar do aparelho, e na tela principal não aparecer item nenhum; a outra situação é você cadastrar um item e clicar no botão voltar da barra de superior do aplicativo, desta forma os itens cadastrados aparecerão normalmente.

Mas por que isso acontece? Tem diferença entre o botão voltar do aparelho e o botão voltar da barra superior do aplicativo? A resposta é sim, tem diferença!

A diferença é que, quando se volta através do botão do aparelho, a Activity anterior não é recriada, ou seja, o método onCreate não é executado. Dessa forma, ela simplesmente carrega a Activity do jeito como ela estava na memória. No entanto, quando voltamos através do botão superior, a Activity é recriada e o método onCreate é chamado. Sendo assim, agora entendemos por que quando voltamos com o botão do aparelho não aparecem os itens cadastrados - porque o código que preenche a lista está no onCreate, assim ele não é chamado.

Vamos resolver isso transferindo o código que preenche a lista para o método onResume, que é chamado sempre quando uma tela for aparecer, independente se ela está sendo criada ou se simplesmente está voltando.

Vamos então implementar o método onResume na classe MainActivity e transferir a linha de código produtosAdapter.addAll(produtosGlobal) para lá.

```
override fun onResume() {
  super.onResume()

  produtosAdapter.addAll(produtosGlobal)
}
```

Mas tem um problema: você verá que a variável produtosAdapter não existe nesse método porque ela foi criada no método onCreate. E está certo manter a criação do adaptador lá no onCreate pois ele será criado uma única vez e depois só vamos atualizar seus itens. Mas para resolver nosso problema podemos acessar o produtosAdapter através da própria list_view_produtos:

```
override fun onResume() {
  super.onResume()

  val adapter = list_view_produtos.adapter as ProdutoAdapter

  adapter.addAll(produtosGlobal)
}
```

Ótimo! Teste o aplicativo novamente e veja se está tudo funcionando.

Se você cadastrou alguns itens, voltou para a tela principal, cadastrou mais itens, você percebeu que temos mais um pequeno problema a resolver. Da forma como está, você deve ter visto que algumas vezes a lista exibe itens duplicados. Isso está acontecendo porque o método adapter.addAll(produtosGlobal) sempre adiciona todos os itens na lista, não considerando possibilidade de ela já possuí-los, e por isso ficam itens duplicados. Para resolver isso, vamos limpar a lista antes de adicionar itens novos com o método clear() do adaptador. Este método simplesmente limpa a lista.

```
override fun onResume() {
  super.onResume()

  val adapter = list_view_produtos.adapter as ProdutoAdapter
  adapter.clear()
  adapter.addAll(produtosGlobal)
}
```

Agora sim temos o funcionamento correto da lista! Faça alguns testes, veja se agora os itens param de duplicar.

Somando os itens da lista

Agora que já temos o fluxo básico funcionando, conseguimos cadastrar um item e conseguimos visualizá-lo na lista. Vamos implementar a funcionalidade de somatória na tela principal. No layout pensado inicialmente, existe uma TextView para exibir a soma dos itens cadastrados na lista. Veja novamente o layout:

Figura 14: Layout da lista.

No layout que criamos lá no começo, já definimos essa TextView com id txt_total, então precisamos somente somar os itens da lista e atualizar esse TextView. Para fazer essa soma, na maioria das linguagens precisaríamos fazer uma estrutura de repetição e, dentro dela, ir acumulando o valor de cada item. Em Kotlin poderíamos fazer assim também, o código seria o seguinte:

```
var soma = 0.0

for ( item in produtosGlobal) {
  soma += item.valor * item.quantidade
}
```

Não é um código complicado, simplesmente aplicamos um for na variável produtosGlobal e, a cada item, fazemos a multiplicação do valor pela quantidade e acumulamos na variável soma. Mas podemos deixá-lo mais simplificado utilizando a função sumByDouble da lista. Através dessa função, podemos fazer uma soma de todos os itens da lista sem precisar de uma estrutura de repetição. Veja como ficaria o código:

```
val soma = produtosGlobal.sumByDouble { it.valor * it.quantidade }
```

E pronto! Somente essa linha substitui aquele for, então vamos utilizar desta forma. Tendo a variável soma, só precisamos atualizar a TextView, e já vamos aproveitar para formatar essa variável no formato da moeda:

```
val f = NumberFormat.getCurrencyInstance(Locale("pt", "br"))
txt_total.text = "TOTAL: ${ f.format(soma)}"
```

Faça os imports das classes NumberFormat:

```
import java.text.NumberFormat
```

E Locale:

```
import java.util.Locale
```

O código completo do método onResume:

```
override fun onResume() {
  super.onResume()

  val adapter = list_view_produtos.adapter as ProdutoAdapter

  adapter.clear()
  adapter.addAll(produtosGlobal)

  val soma = produtosGlobal.sumByDouble { it.valor * it.quantidade }
  val f = NumberFormat.getCurrencyInstance(Locale("pt", "br"))

  txt_total.text = "TOTAL: ${ f.format(soma)}"
}
```

Teste o aplicativo e veja se a soma está sendo feita corretamente!

Acessando a galeria de imagens

Vamos resolver agora a última parte do nosso aplicativo, que é poder incluir uma imagem para o produto cadastrado. Vamos fazer isso acessando a galeria de imagens do celular e definindo a foto de acordo com a imagem escolhida pelo usuário.

Para acessar a galeria de imagens do aparelho, precisamos criar um objeto de intenção Intent e definir alguns parâmetros para ele. Mas, antes de começar o código em si, vamos entender primeiro o que é um objeto Intent e como ele funciona.

O objeto Intent

A Intent é um objeto de mensagem que pode ser usado para solicitar uma ação de outro componente de aplicativo. Embora os intents facilitem a comunicação entre componentes de diversos modos, vamos ver como uma Intent trabalha para iniciar uma Activity:

Iniciar uma atividade:

Uma Activity representa uma única tela em um aplicativo. Como já visto, é possível iniciar uma nova instância de uma Activity passando uma Intent para o método startActivity(). A Intent descreve a atividade a iniciar e carrega todos os dados necessários.

Veja um exemplo de uma Intent para uma chamada telefônica:

```
//definindo a intent com ação para chamada telefônica
val chamada = Intent(Intent.ACTION_DIAL)

//definindo o parâmetro que a intent usará para chamada
chamada.setData("tel:1155559900");

//inicializando a activity de chamada telefônica
startActivity(chamada);
```

Esse código descreve como efetuar uma chamada telefônica a partir de uma Intent. Esse tipo de solicitação é chamado de Intent implícita pois não estamos especificando exatamente qual Activity o Android vai abrir para efetuar a chamada, simplesmente estamos definindo a ação ACTION_DIAL e o sistema operacional resolverá isso para nós.

Em alguns casos, desejamos receber algum retorno da Activity que foi iniciada. Nosso caso será um exemplo disso: precisamos inicializar a Activity da galeria para o usuário escolher uma imagem; assim que o usuário tiver escolhido, precisamos que essa imagem seja enviada como resposta para o aplicativo, para podermos salvá-la.

Para esses casos, a inicialização da Activity deve ser feita por meio do método startActivityForResult() e para capturar o resultado devemos implementar o método onActivityResult. Veja um exemplo de como esses métodos trabalham em conjunto:

```
fun abrirGaleria() {
  //definindo a ação de conteúdo
  val intent = Intent(Intent.ACTION_GET_CONTENT)

  //definindo filtro para imagens

  intent.type = "image/*"

  //inicializando a activity com resultado
  startActivityForResult(Intent.createChooser(intent, "Selecione uma imagem"), COD_IMAGE)
}
```

Faça o import da classe Intent:

```
import android.content.Intent
```

> *Atenção: esse código não fica dentro do método onCreate, é uma função separada, mas ainda no escopo da Activity.*

O código anterior demonstra uma chamada ao método startActivityForResult para abrir a galeria de imagens do celular. Observe que, junto à chamada do método, é necessário passar um número inteiro que foi definido

na variável COD_IMAGE. Esse número é de escolha do desenvolvedor, não existe um número predefinido. Esse código será usado no método onActivityResult para saber que a resposta veio desta requisição.

Crie também a variável COD_IMAGE no escopo da Activity, geralmente no começo da classe, antes do método onCreate. Vou definir um valor de 1000, mas como disse, esse valor é de escolha do desenvolvedor. Veja como isso fica no código:

```
class CadastroActivity: AppCompatActivity() {

  val COD_IMAGE = 1000

 ...

  //restante do código
```

Veja agora a implementação do método onActivityResult:

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
  super.onActivityResult(requestCode, resultCode, data)

  if (requestCode == COD_IMAGE && resultCode == Activity.RESULT_OK) {

    if (data != null) {

      //Neste ponto podemos acessar a imagem escolhida através da variável "data"
    }
  }
}
```

Aqui você precisará do import da classe Activity:

```
import android.app.Activity
```

O que precisamos é exatamente disso! Abra o arquivo CadastroActivity e implemente as funções abrirGaleria e onActivityResult assim como as definimos.

Em seguida, vamos definir que a função abrirGaleria seja chamada quando o usuário clicar no objeto ImageView, para isso inclua o listener de clique do ImageView para chamar a função abrirGaleria no método onCreate da Activity:

```
//definindo o ouvinte da foto do produto
img_foto_produto.setOnClickListener { abrirGaleria() }
```

Aqui você já consegue testar o aplicativo. Você verá que conseguimos abrir a galeria e selecionar uma imagem, mas ela ainda não aparece no nosso aplicativo. Vamos continuar o código para trazer a imagem escolhida para dentro do App.

O método onActivityResult nos dá acesso a 3 variáveis: requestCode, resultCode e data?. O requestCode é o código da requisição, um número inteiro pelo qual podemos diferenciar as requisições feitas. O resultCode é o código do resultado da Activity, ele pode ter o conteúdo igual a Activity.RESULT_OK, o que significa que o usuário não cancelou a ação, ou pode ser igual a Activity.RESULT_CANCELED, o que significa que o usuário cancelou a ação. Por fim, temos a variável data?, que pode ser nula, então devemos sempre fazer a verificação de nulo nela. Ela carrega o conteúdo que foi retornado da Activity, no nosso caso, ela vai carregar a URI da imagem.

Como precisamos guardar essa imagem em um Bitmap, podemos ler essa URI e transformar para Bitmap. Já vamos aproveitar e exibir a imagem na tela, isso será feito com o código a seguir:

```
//lendo a URI com a imagem
val inputStream = contentResolver.openInputStream(data.getData()!!);

//transformando o resultado em bitmap
imageBitMap = BitmapFactory.decodeStream(inputStream)

//Exibir a imagem no aplicativo
img_foto_produto.setImageBitmap(imageBitMap)
```

Vamos então implementar esse código no nosso aplicativo. Primeiro, defina a variável imageBitMap no escopo da classe CadastroActivity:

```
var imageBitMap: Bitmap? = null
```

Você precisará importar a classe Bitmap:

```
import android.graphics.Bitmap
```

Agora complete o código do método onActivityResult:

```
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
  super.onActivityResult(requestCode, resultCode, data)

  if (requestCode == COD_IMAGE && resultCode == Activity.RESULT_OK) {

    if (data != null) {

      //lendo a uri com a imagem
      val inputStream = contentResolver.openInputStream(data.getData()!!);

      //transformando o resultado em bitmap
      imageBitMap = BitmapFactory.decodeStream(inputStream)
```

```
      //Exibir a imagem no aplicativo
      img_foto_produto.setImageBitmap(imageBitMap)
    }

  }
}
```

Faça também o import da classe BitmapFactory:

```
import android.graphics.BitmapFactory
```

Agora você pode testar o aplicativo novamente e você verá que já conseguimos procurar uma imagem e ela é exibida no aplicativo! Mas ainda falta um detalhe: se você salvar um item, você vai ver que a imagem ainda não está sendo salva, isso porque não estamos salvando a variável imageBitMap. Vamos fazer isso.

Localize no método onCreate a seguinte linha de código:

```
val prod = Produto(produto, qtd.toInt(), valor.toDouble())
```

Essa linha monta o objeto que está sendo salvo. Vamos modificá-la para incluir a variável imageBitMap:

```
val prod = Produto(produto, qtd.toInt(), valor.toDouble(), imageBitMap)
```

E pronto! Agora você pode testar o aplicativo e ver o fluxo completo em funcionamento.

Resumo

Neste projeto, vimos alguns recursos muito importantes do Android e muitas coisas legais da linguagem Kotlin. Vimos como montar uma lista personalizada com um adaptador customizado, trabalhar com mais de uma Activity, como podemos simular um processo de cadastro, acessar a galeria de imagens e muito mais!

No próximo projeto, vamos construir um aplicativo utilizando banco de dados, assim conseguiremos guardar informações no disco para que não se percam quando o aplicativo for fechado, ou até mesmo quando o telefone for desligado. Espero você no próximo projeto.