

**CURSOS  
TÉCNICOS**

**DESENVOLVIMENTO DE  
APLICATIVOS I**

Eixo Informática para Internet

**UNIDADE 6**

## SUMÁRIO

### UNIDADE 6

1. TRABALHANDO EM UM APP DINÂMICO .....	3
1.1. O que vamos aprender? .....	3
1.2. Obtendo o projeto inicial .....	3
1.3. Criando classes e packages .....	4
1.4. Data Class .....	6
1.5. Adicionando um Card no App .....	8
1.6. Criando a lista de cards .....	12
2. Criando um App simples com grade (grid) .....	14
2.1 Obtendo os recursos para o projeto .....	14
2.2 Criando o projeto .....	14
2.3 Definindo o layout com a grid .....	16
3. Referências .....	20

## **UNIDADE 6**

### **1. TRABALHANDO EM UM APP DINÂMICO**

Na unidade anterior, trabalhamos com vários tópicos importantes para o desenvolvimento de um aplicativo, criamos uma calculadora básica, para o cálculo de gorjetas. Nesta próxima unidade, vamos começar com um aplicativo mais interativo, onde aplicamos uma lista visual usando imagens e elementos roláveis.

#### **1.1. O que vamos aprender?**

Vamos trabalhar com uma base de app com imagens e textos, que mostra uma lista de frases inspiradoras combinadas com belas imagens para melhorar seu dia. Os tópicos são:

- ✓ Como criar um **card do Material Design** usando o Jetpack Compose.
- ✓ Como criar uma **lista rolável usando o Jetpack Compose**.
- ✓ **LazyColumn**.
- ✓ **List**.
- ✓ Trabalhando com **packages** (pacotes).
- ✓ Criando e trabalhando com **classes**.
- ✓ **Data Class**.
- ✓ Trabalhando com **grids**.

#### **1.2. Obtendo o projeto inicial**

Nesta unidade vamos trabalhar com vários conceitos, portanto está sendo disponibilizado um projeto inicial, com um pré código com suas funções.

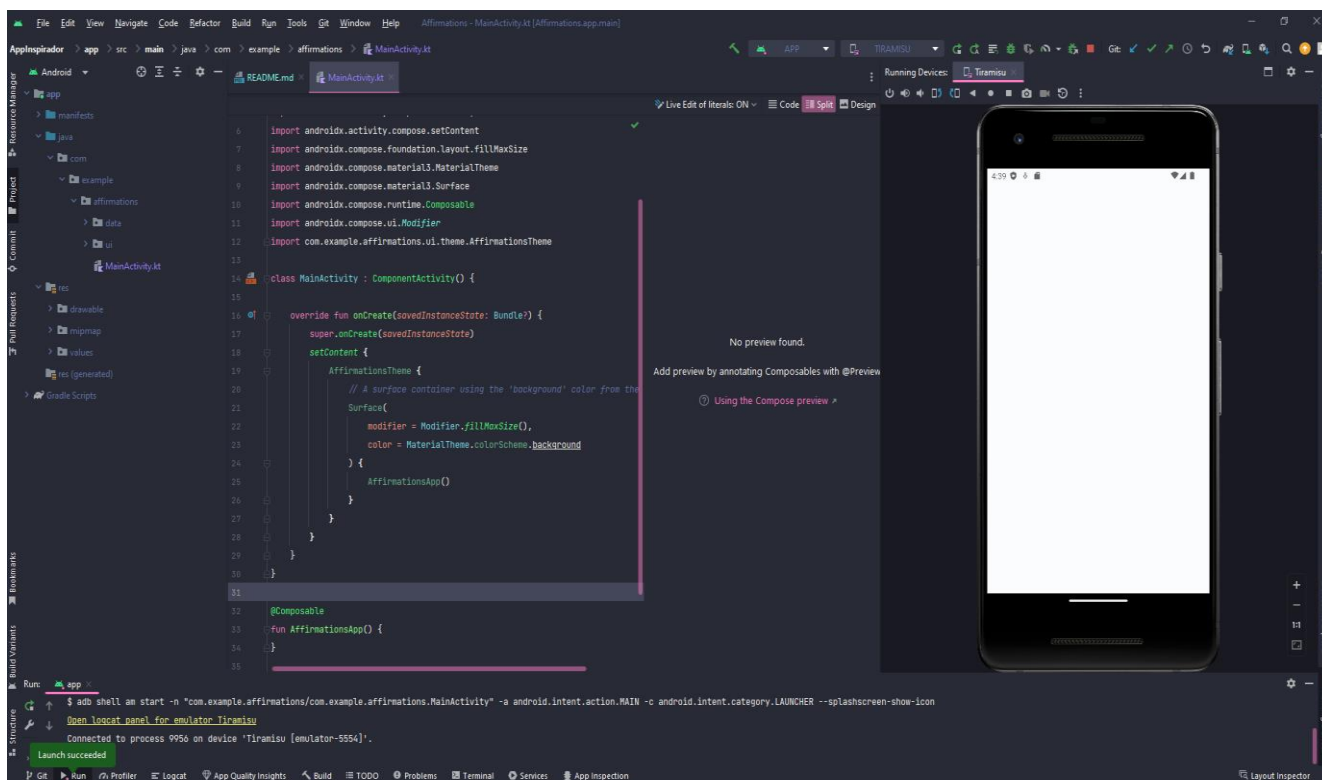
**Lembrando que este código disponibilizado é adaptado da documentação [Desenvolvedores Android](#)** e seu link está em referência no readme do repositório abaixo.

#### **1. Clique no link abaixo:**

Trabalhando com código inicial → neste link:  
<https://github.com/cristijung/AppInspirador/tree/inicio>

## DESENVOLVIMENTO DE APLICATIVOS I

2. No repositório faça download compactado dele, como foi feito na unidade anterior.
3. Descompacte a pasta e abra o projeto no Android Studio.
4. Execute o emulador do projeto, você verá uma tela em branco a princípio.



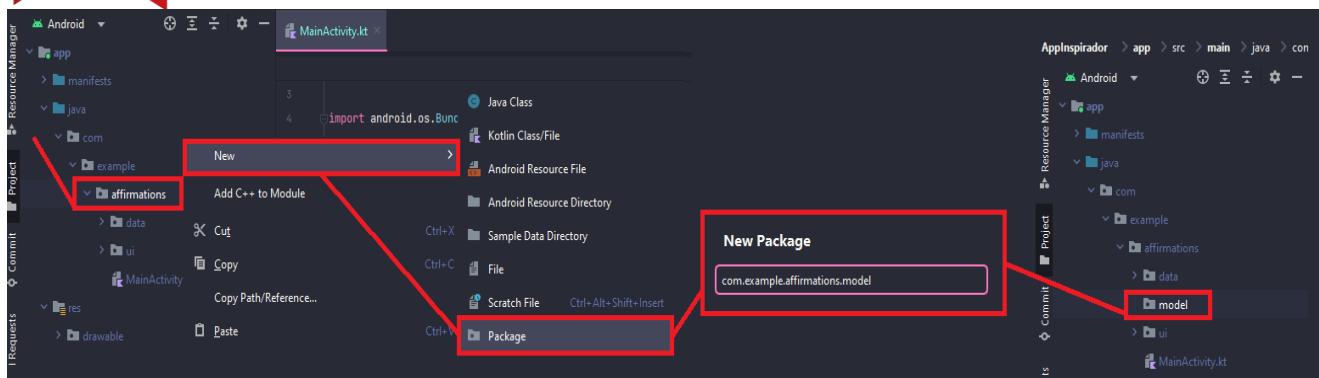
Fonte da imagem: autoria própria

### 1.3. Criando classes e packages

Em apps Android, as listas são compostas por itens. Para dados únicos, isso pode ser algo simples, como uma string ou um número inteiro. Para itens de lista com vários dados, como imagem e texto, **vamos precisar de uma classe que contenha todas essas propriedades**. As classes de dados são um tipo de classe que contém apenas propriedades, e podem fornecer alguns métodos utilitários para trabalhar com essas propriedades.

Nesta primeira etapa, iremos criar um pacote → observe a próxima imagem:

Vamos dar o nome de 'model' para este novo pacote.



Fonte da imagem: autoria própria

### Mas o que são estas packages?

Packages (pacotes) referem-se a uma maneira de organizar e agrupar classes, interfaces, funções e outros elementos relacionados em uma estrutura hierárquica. Esses pacotes, ajudam a gerenciar a complexidade do código, permitindo que você divida seu projeto em módulos lógicos e reutilizáveis.

**Em outras palavras, os pacotes são usados para agrupar funcionalidades relacionadas e fornecer um mecanismo para evitar conflitos de nomes entre diferentes partes do código.** Isso é especialmente importante, quando estamos desenvolvendo aplicativos maiores e mais complexos, onde várias partes do código podem ter nomes semelhantes.

Vejamos alguns tópicos:

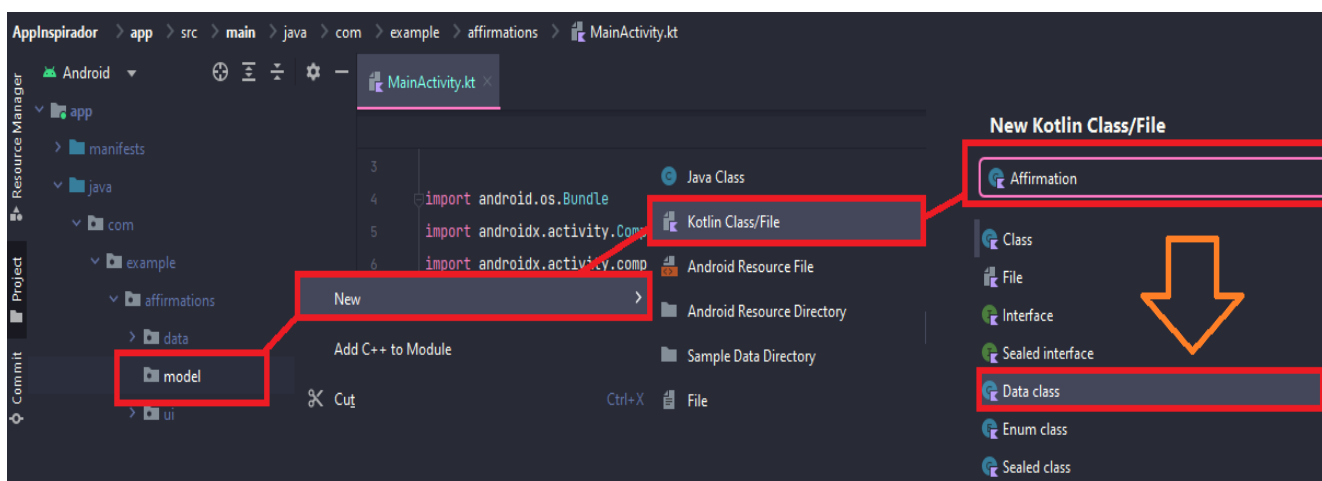
- ✚ **Declaração de pacote:** no início de um arquivo de código-fonte Kotlin, você define a qual pacote o arquivo pertence, usando a declaração **package**.
- ✚ **Hierarquia de pacotes:** podemos organizar pacotes em uma estrutura hierárquica, refletindo a estrutura do seu projeto. Isso ajuda a separar diferentes componentes e funcionalidades.
- ✚ **Acesso a classes e símbolos:** classes e símbolos dentro de um pacote podem ser acessados, a partir de outros arquivos Kotlin no mesmo pacote sem a necessidade de importação. No entanto, para acessar classes e símbolos de outros pacotes, precisamos importá-los usando a declaração **import**.
- ✚ **Visibilidade:** Kotlin fornece modificadores de visibilidade, como **private**, **internal**, **protected** e **public**, para controlar quais partes do código podem acessar elementos dentro de um pacote.

🚦 **Reutilização e modularidade:** organizar seu código em pacotes permite criar módulos reutilizáveis, que podem ser compartilhados entre diferentes partes do aplicativo ou até mesmo entre diferentes aplicativos.

Nosso próximo passo será criar uma classe, observe a próxima imagem:

Nome da classe: **Affirmation**

Tipo: **Data class**




Fonte da imagem: autoria própria

#### 1.4. Data Class

Data class é uma classe especial que é projetada para armazenar dados imutáveis de forma concisa. Ela fornece, automaticamente, uma série de funcionalidades úteis, como a geração automática de métodos comuns como **equals()**, **hashCode()**, **toString()** e **copy()**. As data classes são, frequentemente, usadas para representar objetos simples que carregam dados, como registros, pontos no espaço, informações de usuário, etc.

No nosso projeto, vale salientar aqui que: cada Affirmation consiste em uma imagem e uma string (no projeto temos 9 imagens já baixadas do projeto inicial). Vamos declarar duas variáveis val na classe de dados Affirmation, sendo que uma delas precisa ter o nome **stringResourceId** e a outra, **imageResourceId**, entretanto, ambas precisam ser do tipo números inteiros.

**Observe a imagem da classe/arquivo Affirmation.kt**



```

1 package com.example.affirmations.model
2
3 data class Affirmation(
4     val stringResourceId: Int,
5     val imageResourceId: Int
6 )

```

Fonte da imagem: autoria própria

Próximo passo será associar as notações específicas, para isso iremos anotar a propriedade **stringResourceId** com a anotação **@StringRes** e **imageResourceId** com a anotação **@DrawableRes**.

### IMPORTANTE!

O **stringResourceId** representa um ID para o texto da afirmação armazenado em um recurso de string. O **imageResourceId** representa um ID para a imagem da afirmação armazenada em um recurso drawable, ou seja, na pasta drawable.

### Affirmation.kt



```

1 import androidx.annotation.DrawableRes
2 import androidx.annotation.StringRes
3
4 data class Affirmation(
5     @StringRes val stringResourceId: Int,
6     @DrawableRes val imageResourceId: Int
7 )

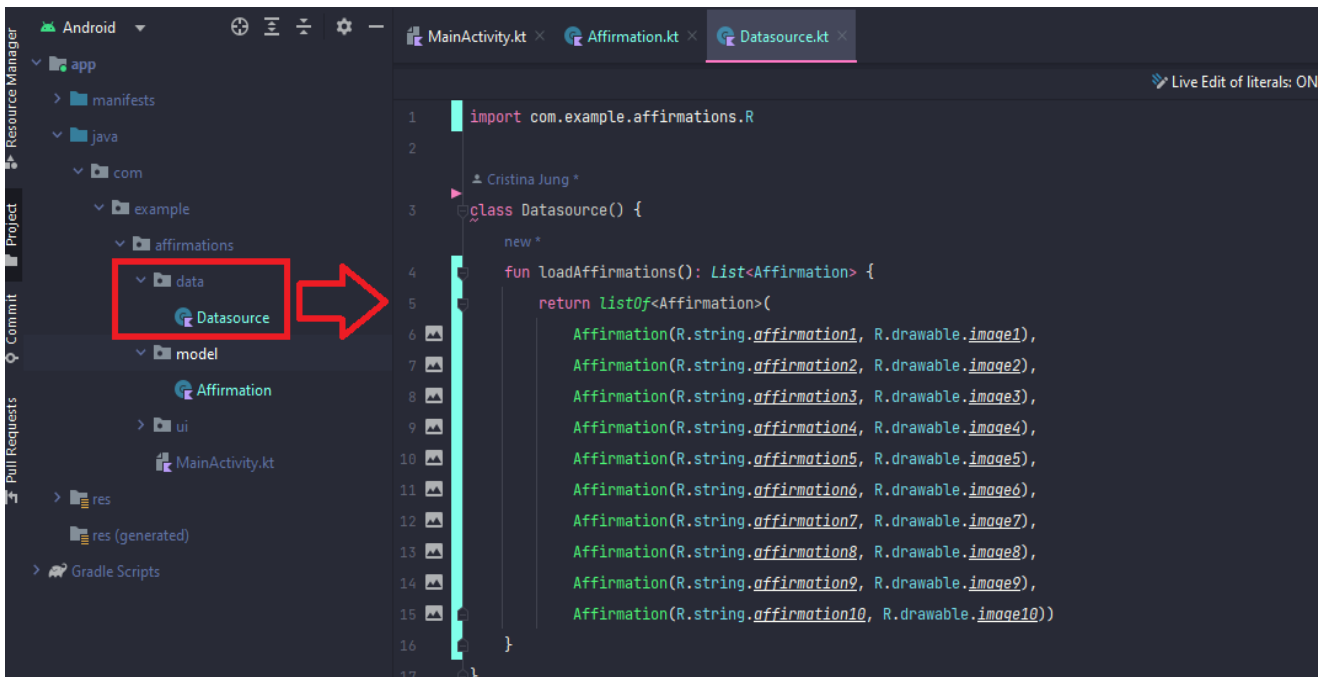
```

Fonte da imagem: autoria própria

No pacote **com.example.affirmations.data**, vamos abrir o arquivo **Datasource.kt** e remover os comentários das instruções de importação e do conteúdo da classe **Datasource**.

## Datasource.kt

As imagens já se encontram no projeto baixado do github.



Fonte da imagem: autoria própria

### Importante!

O **método loadAffirmations()** reúne todos os dados fornecidos no código inicial e os retorna como uma lista. Ele será usado mais tarde para criar a lista rolável.

### 1.5. Adicionando um Card no App

O app precisa exibir uma lista de imagens e textos, esta é a regra de negócio dele. A primeira etapa na configuração da UI para exibir uma lista é criar um item correspondente. Cada item da 'inspiração' é composto por uma imagem e uma string (texto). Os dados de cada um desses itens vêm com o código inicial já disponibilizado no github, e precisaremos criar o componente de UI para mostrar cada um destes itens.

Neste caso específico, o item é um elemento Card combinável, que contém os elementos Image e Text. No Compose, Card é uma plataforma que mostra conteúdo e ações em um único contêiner.

**Um "card" é um componente de interface que geralmente é usado para exibir informações ou conteúdo em uma área retangular com sombras e bordas arredondadas.** É uma maneira popular e com ótima experiência do usuário, usada para apresentar conteúdo de forma organizada e esteticamente agradável.



O Compose simplifica a criação de cartões (cards) por meio do componente Card, que é fornecido como parte da biblioteca Compose. O Card oferece uma abstração para criar essa interface de usuário comum, com sombras e cantos arredondados automaticamente aplicados.

No nosso App, o card mostra uma imagem com texto abaixo. Esse layout vertical pode ser alcançado usando um elemento **Column combinável encapsulado em um Card**.

O card de afirmação ficará assim na visualização:



Fonte da imagem: autoria própria

### **No arquivo MainActivity.ky:**

Abaixo da nossa função **AffirmationsApp()**, vamos criar outra função com nome de **AffirmationCard()** com notação de **@Composable**. Como parâmetros, vamos passar o objeto Affirmation que está vindo lá do pacote model e mais outro parâmetro com valor padrão de Modifier. Quanto ao Modifier, passar o modificador para cada elemento combinável e defini-lo como valor padrão é uma prática recomendada.

- ❖ Em seguida, iremos chamar o elemento Card combinável também passando o parâmetro modifier.
- ❖ Como nosso layout é vertical, nada mais justo do que declararmos o elemento Column combinado ao Card. Os itens em um elemento Column combinável se organizam verticalmente na interface. Isso permite que possamos posicionar uma imagem acima do texto associado.

❖ Vamos agora adicionar um elemento `Image` combinável ao corpo da lambda do `Column`. Não esqueça que uma `Image` combinável sempre exige um recurso para exibição e uma `contentDescription`. O **recurso precisa ser um `painterResource`, transmitido ao parâmetro `painter`. O método `painterResource` carregará drawables vetoriais ou formatos de recursos rasterizados, como PNGs.** Além disso, vamos passar um **`stringResource`** para o parâmetro **`contentDescription`**.

Podemos observar na próxima imagem, que adicionamos um **`contentScale`**, que determina como a imagem deverá ser dimensionada e exibida. No `Modifier`, declaramos `fillMaxWidth`.



```

46  @Composable
47  fun AffirmationCard(affirmation: Affirmation, modifier: Modifier = Modifier) {
48      Card(modifier = modifier) { this: ColumnScope
49          Column { this: ColumnScope
50              Image(
51                  painter = painterResource(affirmation.imageResourceId),
52                  contentDescription = stringResource(affirmation.stringResourceId),
53                  modifier = Modifier
54                      .fillMaxWidth()
55                      .height(194.dp),
56                  contentScale = ContentScale.Crop
57              )
58          }
59      }
60  }
  
```

Fonte da imagem: autoria própria

A imagem já está adicionada e seus recursos virão da pasta `drawable`, agora precisamos declarar o texto. Dentro da `Column`, vamos criar um `Text` combinável após o elemento `Image`. Vamos passar um **`stringResource`** da **`affirmation.stringResourceId`** ao parâmetro `text`, vamos declarar também o objeto `Modifier` com o atributo **`padding` definido como `16.dp`** com a definição de um tema de texto passando **`MaterialTheme.typography.headlineSmall`** ao parâmetro `style`.

```

49  @Composable
50  fun AffirmationCard(affirmation: Affirmation, modifier: Modifier = Modifier) {
51      Card(modifier = modifier) { this: ColumnScope
52          Column { this: ColumnScope
53              Image(
54                  painter = painterResource(affirmation.imageResourceId),
55                  contentDescription = stringResource(affirmation.stringResourceId),
56                  modifier = Modifier
57                      .fillMaxWidth()
58                      .height(194.dp),
59                  contentScale = ContentScale.Crop
60              )
61              Text(
62                  text = LocalContext.current.getString(affirmation.stringResourceId),
63                  modifier = Modifier.padding(16.dp),
64                  style = MaterialTheme.typography.headlineSmall
65              )
66          }
67      }
68  }
    
```

Fonte da imagem: autoria própria

Precisamos agora gerar a visualização do nosso card, para isso, lá no final do arquivo MainActivity, vamos colocar o nosso Preview.

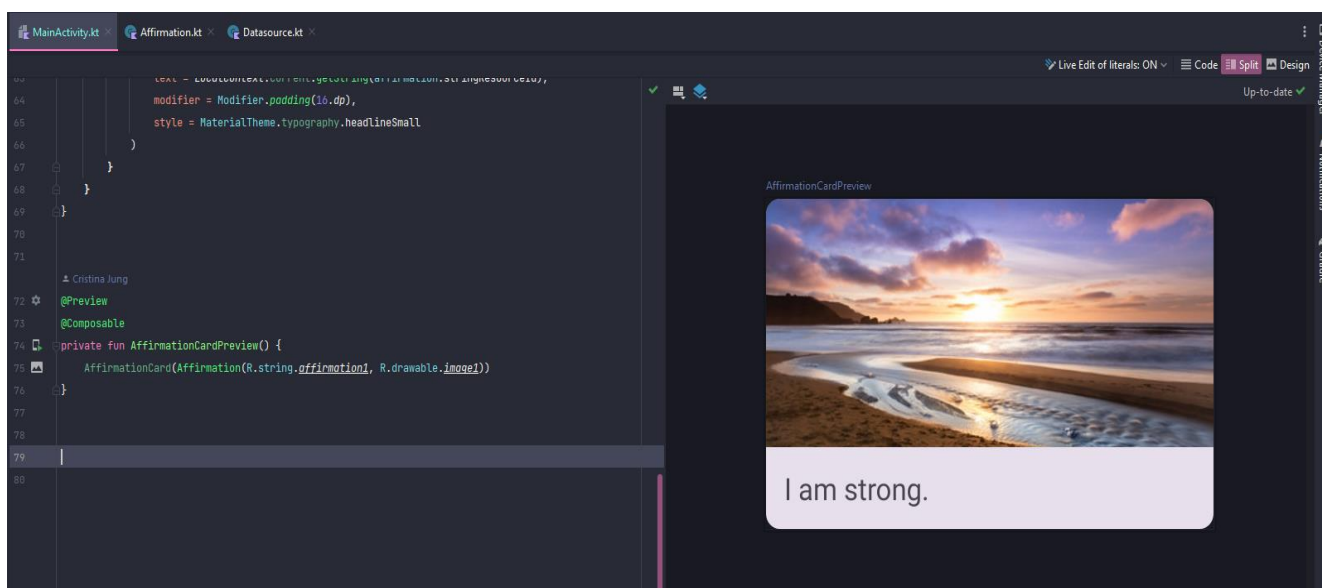
Uma dica, não esqueça de que tudo precisa ser importado para que execute de forma adequada.

```

72  @Preview
73  @Composable
74  private fun AffirmationCardPreview() {
75      AffirmationCard(Affirmation(R.string.affirmation1, R.drawable.image1))
76  }
    
```

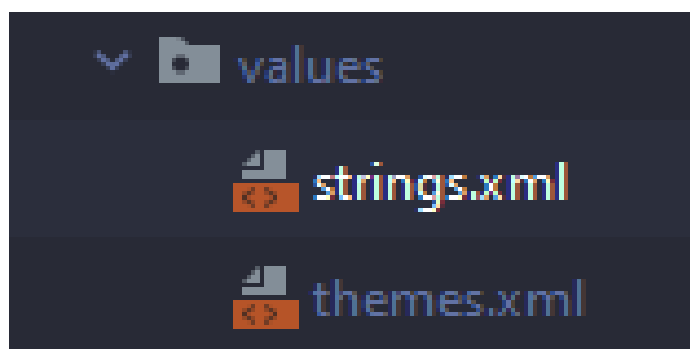
Fonte da imagem: autoria própria

A visualização no modo Design:



Fonte da imagem: autoria própria

Se deseja alterar os textos que são exibidos nesta aplicação, será possível fazer esta edição no arquivo **strings.xml**, que se encontra dentro da pasta 'res':



Fonte da imagem: autoria própria

### **1.6. Criando a lista de cards**

O componente do item é o elemento fundamental da lista. Depois que o item da lista for criado, ele poderá ser usado para criar o componente da lista.

Crie uma função com o nome `AffirmationList()`, adicione a anotação `@Composable` e declare uma `List` de objetos `Affirmation` como um parâmetro na assinatura do método.

Por falar em **List**, temos que uma "list" se refere a um componente que permite exibir uma coleção de itens de maneira rolável e vertical em uma interface de usuário. **É uma forma fundamental de mostrar listas de informações, como contatos, mensagens, feeds de notícias, etc.**

Geralmente quando usamos uma `List`, o Compose já sugere a utilização da **LazyColumn** quando o layout for vertical ou a **LazyRow**, quando o layout for horizontal.

Já o **LazyColumn** é um componente do JetpackCompose para criar listas verticais roláveis de maneira eficiente e responsiva. Ao contrário dos componentes tradicionais de listas em Android, como o RecyclerView, que criam todas as visualizações dos itens de uma vez, a **LazyColumn cria apenas as visualizações dos itens visíveis na tela, economizando recursos e melhorando o desempenho.**

O termo "lazy" (preguiçosa) da LazyColumn **significa que os itens são renderizados apenas quando se tornam visíveis enquanto o usuário rola a lista.** Isso é especialmente útil, quando lidamos com grandes conjuntos de dados, pois evita a carga desnecessária de todos os itens de uma vez, melhorando a velocidade de renderização e economizando memória.

### Declarando a lista:

Vamos criar uma função com o nome **AffirmationList()**, para tanto adicione a anotação **@Composable** e declare uma **List** de objetos **Affirmation** como um **parâmetro no método**. Vamos declarar também um objeto modifier como um parâmetro no método com um valor padrão de Modifier.

Na próxima etapa vamos declarar a lista, porém iremos usar a função LazyColumn. No corpo desta função vamos chamar o método **items()** passando o parâmetro **affirmationList**.

O método **items()** é um construtor fornecido pelo pacote **androidx.compose.foundation.lazy**, que permite criar lista de itens de forma dinâmica em um LazyColumn ou LazyRow. Ele é usado para criar as visualizações dos itens da lista, com base em uma fonte de dados, geralmente uma lista de elementos ou mesmo uma requisição de API.

Observe na próxima imagem como ficou o nosso código:

```

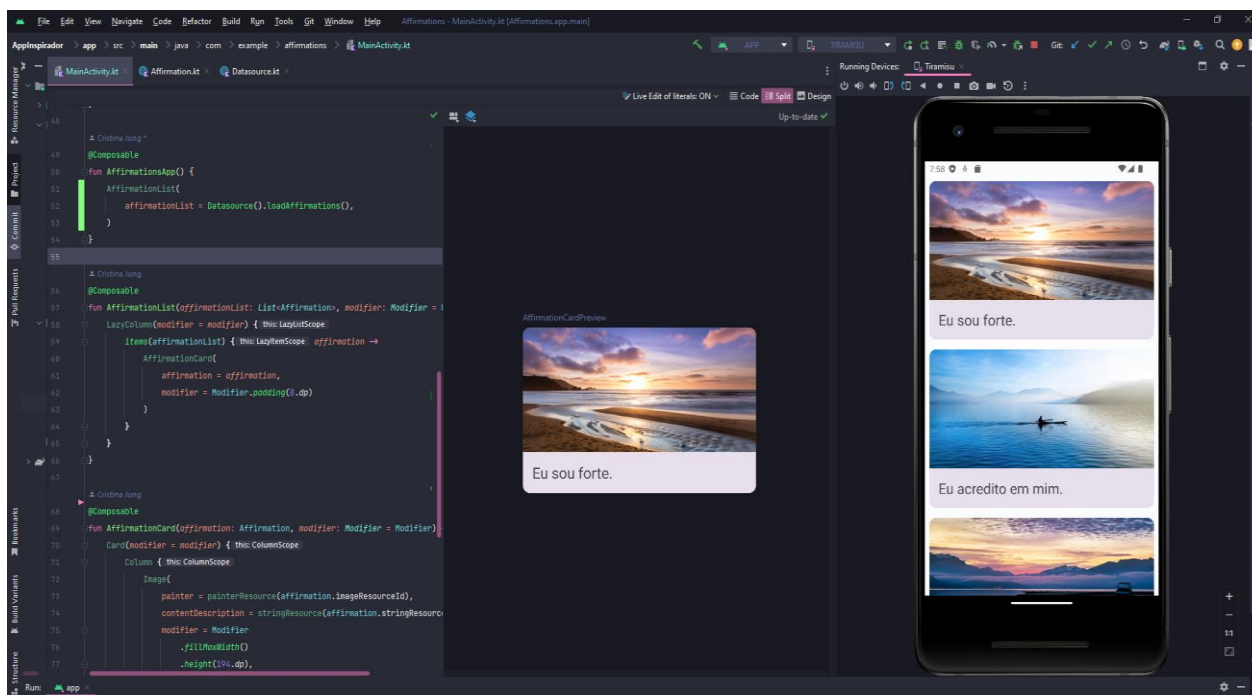
52  @Composable
53  fun AffirmationList(affirmationList: List<Affirmation>, modifier: Modifier = Modifier) {
54    LazyColumn(modifier = modifier) { this: LazyListScope
55      items(affirmationList) { this: LazyItemScope affirmation ->
56        AffirmationCard(
57          affirmation = affirmation,
58          modifier = Modifier.padding(8.dp)
59        )
60      }
61    }
62  }
  
```

Fonte da imagem: autoria própria

Execute o emulador e faça o teste do rolar os cards. Para implementar esta aplicação, você poderá como prática adicionar:

### Prática:

- ✓ Um título no início do App com uma formatação diferenciada (livre)
- ✓ E um breve texto explicando o objetivo deste aplicativo.
- ✓ Cuide para que o layout continue no modo vertical.



Fonte da imagem: autoria própria

## 2. Criando um App simples com grade (grid)

Nesta etapa do livro iremos criar um App básico para organizar em grade os cursos que iremos disponibilizar.

### 2.1 Obtendo os recursos para o projeto

Para dar continuidade a este projeto, será preciso fazer download destes dois recursos:

- [A pasta com as imagens](#)
- [O ícone](#) que irá aparecer em cada imagem de curso

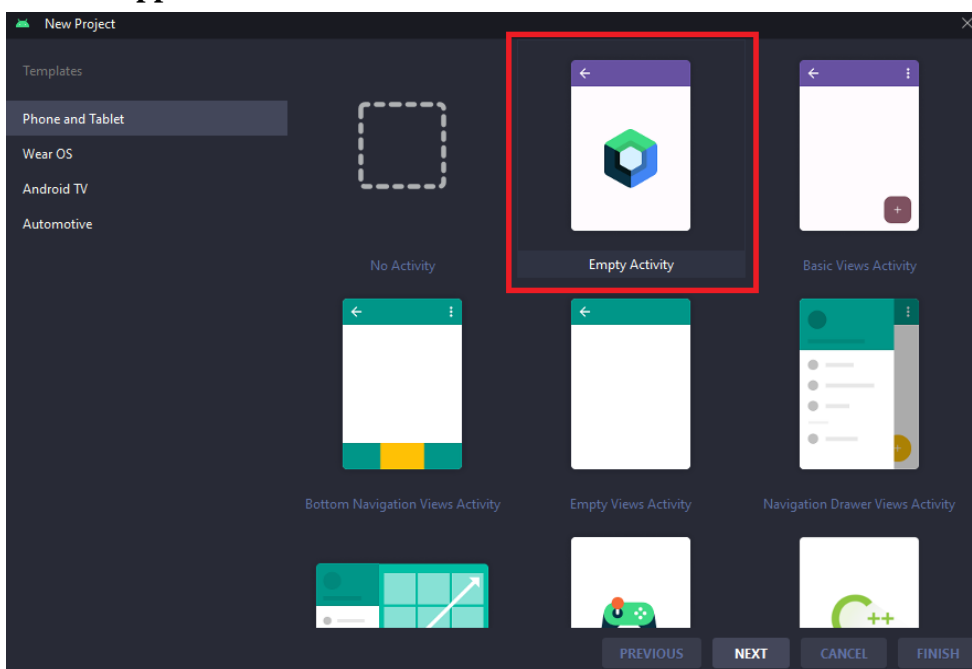
O arquivo de dimensões para as imagens ([dimensions.xml](#)) → deverá ser colocado dentro de: res/values.

### 2.2 Criando o projeto

Vamos criar um novo projeto com esta especificação:

## DESENVOLVIMENTO DE APLICATIVOS I

Nome: **AppCursos**



Fonte da imagem: autoria própria

Na próxima etapa, vamos popular nosso conjunto de dados, para isso, vamos acessar o arquivo strings.xml (pasta res) e inserir os seguintes dados.

### Arquivo strings.xml:

```
<resources>
  <string name="app_name">AppCursos</string>
  <string name="architecture">Arquitetura</string>
  <string name="crafts">Trabalhos Manuais</string>
  <string name="business">Business</string>
  <string name="culinary">Culinária</string>
  <string name="design">Design</string>
  <string name="fashion">Moda</string>
  <string name="film">Filmes</string>
  <string name="gaming">Jogos</string>
  <string name="drawing">Desenho</string>
  <string name="lifestyle">Estilo de Vida</string>
  <string name="music">Música</string>
  <string name="painting">Pintura</string>
  <string name="photography">Fotografia</string>
  <string name="tech">Tecnologia</string>
</resources>
```

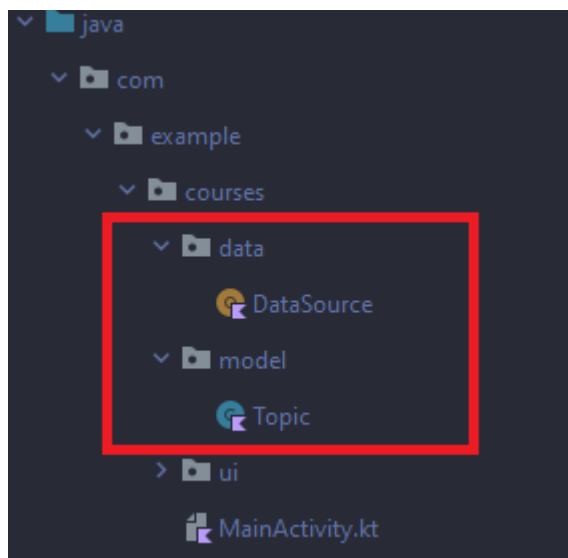
Na próxima etapa, vamos criar a nossa base de dados para que possamos chamar estas informações.

- Vamos criar dois packages, um com nome de model e outro data
- Dentro de model, vamos inserir uma classe chamada Topic.kt
- Dentro de data, vamos declarar a classe DataSource.kt



## DESENVOLVIMENTO DE APLICATIVOS I

Observe as próximas imagens:



Fonte da imagem: autoria própria

### Topic.kt

```

1 package com.example.courses.data
2
3
4 import com.example.courses.R
5 import com.example.courses.model.Topic
6
7 object DataSource {
8     val topics = listOf(
9         Topic("Architecture", availableCourses: 58, R.drawable.architecture),
10        Topic("Crafts", availableCourses: 121, R.drawable.crafts),
11        Topic("Business", availableCourses: 78, R.drawable.business),
12        Topic("Culinary", availableCourses: 118, R.drawable.culinary),
13        Topic("Design", availableCourses: 423, R.drawable.design),
14        Topic("Fashion", availableCourses: 92, R.drawable.fashion),
15        Topic("Film", availableCourses: 165, R.drawable.film),
16        Topic("Gaming", availableCourses: 164, R.drawable.gaming),
17        Topic("Drawing", availableCourses: 326, R.drawable.drawing),
18        Topic("Lifestyle", availableCourses: 305, R.drawable.lifestyle),
19        Topic("Music", availableCourses: 212, R.drawable.music),
20        Topic("Painting", availableCourses: 172, R.drawable.painting),
21        Topic("Photography", availableCourses: 321, R.drawable.photography),
22        Topic("Tech", availableCourses: 118, R.drawable.tech)
23    )
24 }
    
```

### DataSource.kt

```

1
2 package com.example.courses.model
3
4 import androidx.annotation.DrawableRes
5 import androidx.annotation.StringRes
6
7 data class Topic(
8     @StringRes val name: Int,
9     val availableCourses: Int,
10    @DrawableRes val imageRes: Int
11 )
    
```

### 2.3 Definindo o layout com a grid

Agora vamos para o nosso arquivo **MainActivity.kt** montar o nosso layout:

Vamos criar uma função TopicGrid() com a notação de @Composable. Vamos passar o parâmetro Modifier.








```

70  @Composable
71  fun TopicGrid(modifier: Modifier = Modifier) {
72      LazyVerticalGrid(
73          columns = GridCells.Fixed( count: 2),
74          verticalArrangement = Arrangement.spacedBy(dimensionResource(R.dimen.padding_small)),
75          horizontalArrangement = Arrangement.spacedBy(dimensionResource(R.dimen.padding_small)),
76          modifier = modifier
77      ) { this: LazyGridScope
78          items(DataSource.topics) { this: LazyGridItemScope topic →
79              TopicCard(topic)
80          }
81      }
82  }
  
```

Fonte da imagem: autoria própria

### Análise do código acima:

- 
**fun TopicGrid(modifier: Modifier = Modifier):** esta é a definição da função TopicGrid. Ela é uma função com um parâmetro chamado modifier do tipo Modifier com um valor padrão de Modifier. O tipo Modifier é usado para aplicar modificações aos elementos da interface do usuário.
- 
**LazyVerticalGrid:** aqui, estamos usando a função LazyVerticalGrid para criar um grid vertical. A função LazyVerticalGrid é parte da biblioteca de composição e é usada para criar listas ou grids que só criam e renderizam os itens visíveis na tela, o que melhora o desempenho.
- 
**columns = GridCells.Fixed(2),:** definindo o número de colunas no grid. Nesse caso, está usando GridCells.Fixed(2) para criar um grid com 2 colunas.
- 
**verticalArrangement = Arrangement.spacedBy(dimensionResource (R.dimen. padding\_small)),:** definindo o espaçamento vertical entre os itens no grid usando a enumeração Arrangement.spacedBy. O valor do espaçamento é obtido de um recurso de dimensão chamado R.dimen.padding\_small.
- 
**horizontalArrangement = Arrangement.spacedBy(dimensionResource(R.dimen. padding\_small)),:** isso é semelhante à linha anterior, mas está definindo o espaçamento horizontal entre os itens.

## DESENVOLVIMENTO DE APLICATIVOS I

- ✚ **modifier = modifier:** estamos aplicando o modificador modifier que foi passado como parâmetro para a função TopicGrid. Isso permite que quem chama essa função aplique modificações específicas.
- ✚ **items(DataSource.topics) { topic ->:** estamos usando a função items para iterar sobre uma lista de tópicos obtidos de DataSource.topics. Cada topic é passado para a próxima linha como um parâmetro.
- ✚ **TopicCard(topic):** chamando a função TopicCard para criar um cartão de tópico com base no tópico atual da iteração.

A próxima função que iremos criar, será o card, neste caso, o nome: **TopicCard()**. Observe que estamos usando várias propriedades e sobrecargas que se referem à estilização do módulo.

```

@Composable
fun TopicCard(topic: Topic, modifier: Modifier = Modifier) {
    Card {
        Row {
            Box {
                Image(
                    painter = painterResource(id = topic.imageRes),
                    contentDescription = null,
                    modifier = modifier
                        .size(width = 68.dp, height = 68.dp)
                        .aspectRatio(1f),
                    contentScale = ContentScale.Crop
                )
            }

            Column {
                Text(
                    text = stringResource(id = topic.name),
                    style = MaterialTheme.typography.bodyMedium,
                    modifier = Modifier.padding(
                        start = dimensionResource(R.dimen.padding_medium),
                        top = dimensionResource(R.dimen.padding_medium),
                        end = dimensionResource(R.dimen.padding_medium),
                        bottom = dimensionResource(R.dimen.padding_small)
                    )
                )

                Row(verticalAlignment = Alignment.CenterVertically) {
                    Icon(
                        painter = painterResource(R.drawable.ic_grain),
                        contentDescription = null,
                        modifier = Modifier
                            .padding(start = dimensionResource(R.dimen.padding_medium))
                    )

                    Text(
                        text = topic.availableCourses.toString(),
                        style = MaterialTheme.typography.labelMedium,
                        modifier = Modifier.padding(start =
dimensionResource(R.dimen.padding_small))
                    )
                }
            }
        }
    }
}

```

E por fim, estamos gerando a visualização da aplicação:

```

130  @Preview(showBackground = true)
131  @Composable
132  fun TopicPreview() {
133      CoursesTheme {
134          val topic = Topic(R.string.photography, availableCourses: 321, R.drawable.photography)
135          Column(
136              modifier = Modifier.fillMaxSize(),
137              verticalArrangement = Arrangement.Center,
138              horizontalAlignment = Alignment.CenterHorizontally
139          ) { this: ColumnScope
140              TopicCard(topic = topic)
141          }
142      }
143  }
    
```

Fonte da imagem: autoria própria

Retornando à parte comum da MainActivity, onde declaramos o onCreate() e setContent(), vamos verificar o código:

```

51  class MainActivity : ComponentActivity() {
52      override fun onCreate(savedInstanceState: Bundle?) {
53          super.onCreate(savedInstanceState)
54          setContent {
55              CoursesTheme {
56                  // A surface container using the 'background' color from the theme
57                  Surface(
58                      modifier = Modifier.fillMaxSize(),
59                      color = MaterialTheme.colorScheme.background
60                  ) {
61                      TopicGrid(
62                          modifier = Modifier.padding(dimensionResource(R.dimen.padding_small))
63                      )
64                  }
65              }
66          }
67      }
68  }
    
```

Fonte da imagem: autoria própria

### Links para correção dos códigos:

App 1 → <https://github.com/cristijung/AppInspirador.git>

App II → <https://github.com/cristijung/AppCursos.git>

### **3. Referências**

- 
- Múltiplos autores: **KOTLIN DOCS**. 2023. Disponível em: <<https://kotlinlang.org/docs/home.html>>. Acesso em: 17 de julho de 2023
- Múltiplos autores: **JETBRAINS**. 2022. Disponível em: <<https://www.jetbrains.com/pt-br/>>. Acesso em: 18 de julho de 2023
- Múltiplos autores: **DOCUMENTAÇÃO ANDROID STUDIO**. 2023. Disponível em: <<https://developer.android.com/studio>>. Acesso em: 18 de julho de 2023
- Múltiplos autores: ESTADO E JETPACK COMPOSE. 2023. Disponível em: <[Estado e Jetpack Compose | Android Developers](#)>. Acesso em: 13 de agosto de 2023
- Múltiplos autores: **KOTLIN DOCS. ELVIS OPERATOR**. 2023. Disponível em: <<https://kotlinlang.org/docs/home.html>>. Acesso em: 13 de julho de 2023