Capítulo 2

Desenvolvendo Aplicativos Android usando Kotlin

Larissa Cardoso Zimmermann[1], Juliana Martins Leônico Eusébio[1], Maria da Graça Campos Pimentel[1]

[1] Universidade de São Paulo. Instituto de Ciências Matemáticas e de Computação.

larissa.zimmermann@usp.br, julianaleoncio@usp.br, mgp@icmc.usp.br

Abstrat

In this mini-course, the development of applications for Android platform using the Kotlin programming language will be introduced. Essential concepts for native software development for Android platform will be presented concurrently with practical activities. Participants of the mini-course should have access to computers with Android Studio installed and prior knowledge of some programming language. Firstly, the creation of a "Hello World" application will be introduced to familiarize participants with Android Studio. Secondly, the Android Studio Layout Editor will be presented, briefly teaching how to create interfaces with user inputs and buttons. Following that, the concept of the lifecycle of Activities and Fragments will be introduced, aiming to learn how to manage lifecycle events with more organized and maintainable code. Finally, with the goal of managing data in the face of configuration changes of applications for Android, the architecture components ViewModel and LiveData will be taught. These components will be linked to data in order to simplify layout views and eliminate the need for click handlers in the interface controllers. Therefore, by the end of this mini-course, participants will be capable of creating applications for the Android platform with a simplified architecture, configuring interface elements, and maintaining data persistence during application configuration changes.

Resumo

Neste minicurso de tratamento prático, será introduzido o desenvolvimento de aplicativos para a plataforma Android utilizando a linguagem de programação Kotlin. Serão apresentados conceitos essenciais para o desenvolvimento de software nativo para a plataforma Android concomitante com atividades práticas. Para isso, os participantes do minicurso deverão ter acesso a computadores com a plataforma Android Studio instalada e conhecimento prévio de alguma linguagem de programação. Em primeiro lugar, será introduzido como criar um aplicativo "Hello World", com o objetivo de familiarizar os participantes com a plataforma Android Studio. Em segundo lugar, apresentaremos o Editor de Layouts do Android Studio, com o objetivo de ensinar brevemente como criar interfaces, com entradas dos usuários e botões. Em sequência, o conceito de ciclo de vida de Atividades e de Fragmentos será apresentado, com a finalidade de aprender a gerenciar eventos do ciclo de vida com códigos mais organizados e fáceis de manter. Por fim, com o objetivo de gerenciar os dados mediante as mudanças de configuração de aplicativos para plataforma Android, serão ensinados os componentes de arquitetura ViewModel e LiveData. Estes componentes serão vinculados aos dados para que as visualizações do layout sejam simplificadas e elimine a necessidade de gerenciadores de cliques nos controladores da interface. Portanto, ao final deste minicurso, os participantes estarão aptos a criar aplicativos para a plataforma Android, com arquitetura simplificada, configurando elementos de interface e mantendo a persistência dos dados durante mudanças de configurações do aplicativo.

2.1. Introdução

A introdução ao minicurso de Android com Kotlin representa a porta de entrada para o desenvolvimento de aplicativos móveis e para o mundo da computação ubíqua. Em um cenário tecnológico em constante evolução, a criação de aplicativos para a plataforma Android se destaca como uma habilidade essencial. Este curso propõe-se a oferecer uma compreensão prática e abrangente, capacitando os participantes a dominar as nuances do desenvolvimento de *software* para dispositivos Android utilizando a linguagem de programação Kotlin.

A escolha do Android como plataforma é motivada pela sua presença ubíqua em dispositivos móveis, abrangendo desde *smartphones* até *tablets*, *smartwatches* e TVs. A capacidade de criar aplicativos para essa plataforma oferece não apenas uma oportunidade de inovação, mas também a chance de impactar a vida cotidiana dos usuários em escala global.

O minicurso inicia-se com a configuração do ambiente de desenvolvimento no Android Studio, a ferramenta official do Google para criação de aplicativos Android. Os participantes serão guiados passo a passo, desde a instalação até a compreensão das principais funcionalidades, preparando o terreno para uma jornada prática e imersiva.

O marco inicial, a criação do aplicativo "*Hello World*", não é apenas uma formalidade; é um convite para mergulhar no código, compreender a estrutura básica e, mais importante, sentir a empolgação de dar vida a um aplicativo funcional.

A exploração do Editor de *Layouts* do Android Studio oferece a oportunidade de aprimorar a estética e a usabilidade dos aplicativos, garantindo que os desenvolvedores possam criar interfaces atraentes e intuitivas.

À medida que avançamos, mergulharemos nas complexidades do ciclo de vida das atividades e fragmentos, destacando a importância de gerenciar eventos de maneira eficaz. Esse conhecimento não apenas melhora a estabilidade dos aplicativos, mas também aprimora a experiência do usuário.

A implementação da arquitetura *ViewModel* e *LiveData*, conceitos fundamentais do desenvolvimento Android moderno, proporcionará aos participantes ferramentas poderosas para simplificar o código, gerenciar dados de maneira eficiente e criar aplicativos robustos e escaláveis.

Dessa forma, este minicurso não é apenas uma oportunidade de aprendizado; é um convite para desbravar o excitante campo do desenvolvimento de aplicativos Android com Kotlin, abrindo portas para a inovação e a excelência técnica.

2.2. Motivação

A motivação do curso de Desenvolvimento de Aplicativos Android usando Kotlin é alimentada por uma série de fatores que convergem para criar uma experiência de aprendizado enriquecedora e prática. Este curso busca inspirar e capacitar os participantes a se destacarem no desenvolvimento de aplicativos para a plataforma Android, utilizando a linguagem de programação Kotlin.

- Relevância do Android:

O sistema operacional Android é onipresente em dispositivos móveis em todo o mundo. A capacidade de desenvolver aplicativos para essa plataforma oferece oportunidades significativas, pois os aplicativos Android desempenham um papel vital na vida cotidiana, abrangendo desde comunicação até entretenimento e produtividade.

- Kotlin como liguagem de programação para desenvolvimento de aplicativos Android:

A escolha da linguagem de programação Kotlin destaca-se por sua modernidade, expressividade e preferência no ecossistema Android. A transição para Kotlin é uma resposta à demanda crescente por uma linguagem mais concisa e segura, proporcionando aos participantes uma habilidade altamente valorizada no mercado de desenvolvimento de software. Kotlin é a linguagem de programação oficial do Google para o desenvolvimento de aplicativos Android.

- Abordagem Prática:

O curso adota uma abordagem prática desde o início, com a criação do aplicativo "*Hello World*". Essa metodologia permite que os participantes não apenas absorvam conceitos teóricos, mas também coloquem imediatamente em prática o que aprendem, proporcionando uma experiência de aprendizado mais envolvente e eficaz.

- Domínio do Ambiente de Desenvolvimento:

A configuração inicial no Android Studio é uma peça fundamental do curso. Os participantes não apenas aprendem a criar aplicativos, mas também se tornam proficientes no uso da principal ferramenta de desenvolvimento para Android, preparando-os para desafios do mundo real.

- Exploração de Tecnologias Avançadas:

O curso não se limita ao básico; ele avança para conceitos avançados, como o ciclo de vida de atividades e fragmentos, bem como a implementação da arquitetura *ViewModel* e *LiveData*, para persistência e consistência dos dados. Essas tecnologias representam as melhores práticas do desenvolvimento moderno de aplicativos Android.

- Relevância Profissional:

A demanda por desenvolvedores de aplicativos Android qualificados e versados em Kotlin é alta. Este curso visa preparar os participantes não apenas para criar aplicativos, mas também para se destacarem no mercado de trabalho, proporcionando-lhes habilidades práticas e aplicáveis. Portanto, trata-se de um curso extracurricular importante para quem quer se destacar na área de desenvolvimento de aplicativos e computação ubíqua.

Assim, a motivação intrínseca do curso reside na oportunidade de capacitar os participantes a se tornarem a terem uma visão geral do desenvolvimento de aplicativos Android, com conhecimentos práticos e uma base sólida para se aprofudarem na área.

2.3. Público-alvo

O público-alvo deste curso de Android com Kotlin é diversificado, abrangendo desde iniciantes entusiastas da programação até desenvolvedores mais experientes que desejam expandir suas habilidades no desenvolvimento de aplicativos móveis. O curso é projetado para ser inclusivo e oferecer benefícios significativos a diferentes perfis de participantes.

- Iniciantes em Programação:

Este curso é adequado para indivíduos que estão dando os primeiros passos na programação. A abordagem prática, começando com a criação do "*Hello World*", permite uma entrada suave para o mundo da programação e do desenvolvimento de aplicativos.

- Discentes de Ciência da Computação e Engenharia de Software:

Discentes que buscam complementar seus estudos com uma compreensão prática do desenvolvimento de aplicativos para dispositivos móveis. O curso fornece uma base sólida para projetos futuros e experiência relevante para carreiras na área.

- Desenvolvedores Iniciantes em Android:

Profissionais que têm alguma experiência em desenvolvimento, mas desejam migrar ou expandir para o desenvolvimento Android com Kotlin. O curso oferece uma transição

suave, cobrindo conceitos fundamentais e avançados específicos para o ecossistema Android.

- Desenvolvedores Experientes:

Desenvolvedores que já têm experiência em Android, mas desejam atualizar suas habilidades para incluir Kotlin e adotar práticas modernas de arquitetura. O curso oferece uma oportunidade de aprimoramento profissional, explorando conceitos avançados e técnicas atualizadas.

- Pesquisadores e Criadores Independentes:

Pessoas que desejam desenvolver suas próprias ideias de aplicativos ou aprimorar suas habilidades para participar ativamente no desenvolvimento de seus projetos de pesquisa. O curso proporciona conhecimentos práticos para transformar conceitos em aplicativos funcionais.

Em resumo, o curso é projetado para atender a uma ampla gama de participantes, independentemente do nível de experiência em programação ou desenvolvimento Android. A abordagem prática e os conceitos explorados proporcionam uma base sólida para o crescimento profissional e a inovação na área de desenvolvimento de aplicativos móveis.

2.4. Pré-requisitos

Para participar do curso de Desenvolvimento de Aplicativos Android usando Kotlin, os participantes devem atender a alguns pré-requisitos fundamentais para garantir uma experiência de aprendizado eficaz e bem-sucedida. Estes pré-requisitos visam proporcionar uma base mínima de conhecimento e habilidades necessárias para acompanhar o conteúdo do curso:

- Conhecimento Básico em Programação:

Os participantes devem ter conhecimentos básicos em lógica de programação e compreender conceitos fundamentais, como variáveis, estruturas de controle de fluxo (if, else, loops) e estruturas de dados simples.

- Familiaridade com Alguma Linguagem de Programação:

Embora não seja obrigatório ter experiência prévia em Kotlin, é recomendável que os participantes tenham alguma familiaridade com pelo menos uma linguagem de programação, como Java, C++, Python ou JavaScript.

- Ambiente de Desenvolvimento Android Studio Instalado:

Os participantes devem ter o Android Studio instalado em seus computadores antes do início do curso. Instruções sobre como realizar essa instalação podem ser fornecidas antecipadamente.

- Compreensão Básica de Desenvolvimento de Software:

É útil que os participantes tenham uma compreensão básica do ciclo de vida do desenvolvimento de software, incluindo a criação, teste e depuração de aplicativos.

- Disponibilidade de um Dispositivo Android para Testes:

Para obter uma experiência prática completa, os participantes devem ter acesso a um dispositivo Android para testar os aplicativos desenvolvidos durante o curso. Em alternativa, podem utilizar emuladores fornecidos pelo Android Studio.

Estes pré-requisitos são projetados para garantir que os participantes tenham uma base mínima de conhecimentos em programação e estejam prontos para se envolverem nas atividades práticas propostas pelo curso. Aqueles que atendem a esses pré-requisitos estarão mais bem preparados para absorver os conceitos apresentados e aplicá-los no desenvolvimento de aplicativos Android com Kotlin.

2.5. Estrutura em tópicos

O minicurso é organizado em quatro tópicos principais.

1. INTRODUÇÃO AO ANROID STUDIO E CRIAÇÃO DO PRIMEIRO APP

O objetivo desta atividade é ensinar a configurar o Android Studio para usar o Kotlin e a criar um app. Você começará com o “Hello World”, evoluindo até um app que usa arquivos de imagem e um gerenciador de cliques. Você verá como os projetos do Android são estruturados, como usar e modificar visualizações no seu app Kotlin para Android e como garantir que seus apps sejam compatíveis com versões anteriores. Você também aprenderá sobre os níveis de API e as bibliotecas do Android Jetpack.

2. ENTENDENDO O EDITO DE LAYOUT DO ANDROID STUDIO

Nesta etapa, os participantes aprenderão a usar o Editor de Layout do Android Studio para criar layouts lineares e restritos. Os participantes criarão apps que recebem e exibem entradas do usuário, respondem a toques e mudam a visibilidade e a cor das visualizações. Esta lição também ensina a usar a vinculação de dados a elementos de interface.

3. CICLO DE VIDA DE ATIVIDADES E FRAGMENTOS

Nesta fase do minicurso, serão introduzidos os ciclos de vida de atividades e fragmentos. Será ensinado também como gerenciar situações complexas de ciclos de vida. Os participantes trabalharão com um aplicativo com diversos bugs relacionados ao ciclo de vida do Android. Os participantes entenderão como funciona o ciclo de vida de atividades e fragmentos, além de aprender sobre a biblioteca Lifecycle do Android Jetpack, que pode ajudar a gerenciar eventos do ciclo de vida com códigos mais organizados e fáceis de manter.

4. COMPONENTES DE ARQUITETURA

Por fim, será ensinado a usar os objetos ViewModel e LiveData. Os participantes utilizarão objetos ViewModel para autorizar que os dados sobrevivam a mudanças de configuração, como a rotação da tela. Será ensinado como converter os dados da interface de um aplicativo em um LiveData encapsulado e adicionará métodos do observador que são notificados quando o valor do LiveData muda. Será possível também integrará o LiveData e o ViewModel à vinculação de dados para que as visualizações no seu layout se comuniquem diretamente com objetos ViewModel. Essa técnica simplifica o código e elimina a necessidade de gerenciadores de cliques nos controladores da IU.

Este é um minicurso teórico e prático. É seguida a metodologia do Google para a apresentação teórica e o desenvolvimento de aplicativos Android com a ferramenta Android Studio. Os códigos empregados nas atividades foram baseados na documentação oficial de Kotlin para Android fornecido pelo Google.

2.6. Configuração inicial do laboratório

Este minicurso deve ser realizado em laboratório com acesso à Internet e estações de trabalho com o Android Studio instalado. As estações de trabalho devem atender aos requisitos técnicos recomendados pelo distribuidor do Android Studio. A utilização de um dispositivo Android e um cabo de carregamento é opcional, pois estes podem ser substituídos por emuladores de dispositivos Android integrados ao Android Studio.

Na indisponibilidade da alocação de um laboratório, há a possibilidade de uma estrutura alternativa. Os participantes podem usar seus notebooks pessoais para realização das atividades. Neste caso, a sala deve oferecer acesso à internet por sem fio e mesas de trabalho para acomodar os participantes de forma confortável. Vale ressaltar que, dada a duração do treinamento, recomendamos que a sala possua tomadas distribuídas próximo as mesas de trabalho. É recomendado também o participante instalar previamente o Android Studio, pois trata-se de uma aplicação cujo processo de instalação é demorado e não há tempo hábil durante o minicurso.

2.7. Conteúdo do minicurso

O minicurso de “Desenvolvendo Aplicativos *Android* usando Kotlin” proporciona aos participantes uma introdução prática e abrangente ao desenvolvimento de aplicativos para a plataforma *Android*. Durante o curso, os tópicos essenciais são cobertos, incluindo a configuração inicial do ambiente de desenvolvimento no Android Studio, a criação do primeiro aplicativo "*Hello World*", a exploração do Editor de *Layouts*, o entendimento do ciclo de vida de atividades e fragmentos, e a implementação da arquitetura *ViewModel* e *LiveData*.

Os participantes aprendem a configurar corretamente o ambiente de desenvolvimento, tornando-se familiares com o Android Studio e Kotlin. O desenvolvimento prático começa com a criação de um aplicativo simples para estabelecer uma base sólida. Em seguida, o curso explora o Editor de *Layouts*, capacitando os participantes a criar interfaces de usuário eficientes.

A compreensão do ciclo de vida de atividades e fragmentos é destacada, proporcionando aos desenvolvedores as habilidades necessárias para gerenciar eventos de forma eficaz. Além disso, a introdução e implementação dos componentes de arquitetura *ViewModel* e *LiveData* simplificam o desenvolvimento, promovendo código organizado e a gestão eficiente de dados, mesmo em situações de mudanças de configuração do aplicativo.

Ao final do minicurso, os participantes devem estar aptos a desenvolver aplicativos *Android* usando Kotlin, com uma compreensão sólida dos conceitos fundamentais, da criação de interfaces à gestão avançada de dados.

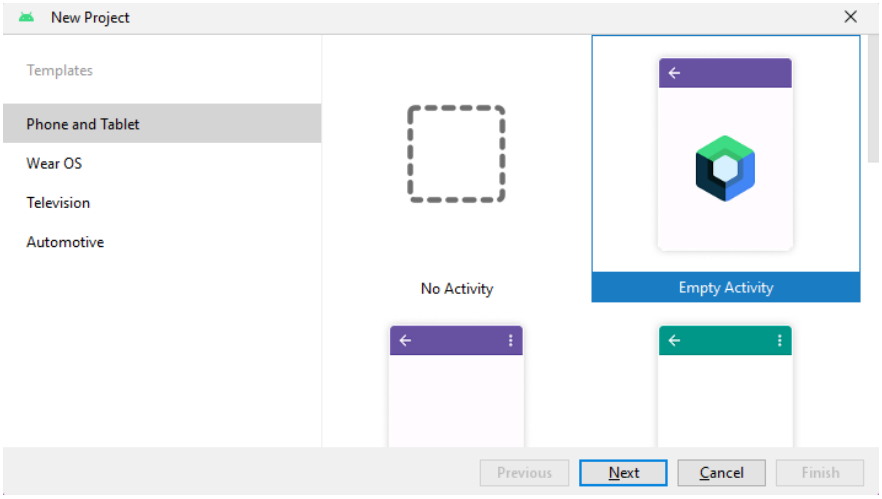
Em seguida, será apresentado o conteúdo do minicurso.

1. Introdução ao Android Studio e criação do primeiro APP

1.1. Hello World!

Primeiro vamos criar um projeto utilizando um modelo pronto do Android Studio:

- Com o Android Studio instalado e aberto, clique em ***New Project***;
- A janela de modelos será aberta. Confira se a guia ***Phone and Tablet*** está selecionada e então escolha o modelo ***Empty Activity***;



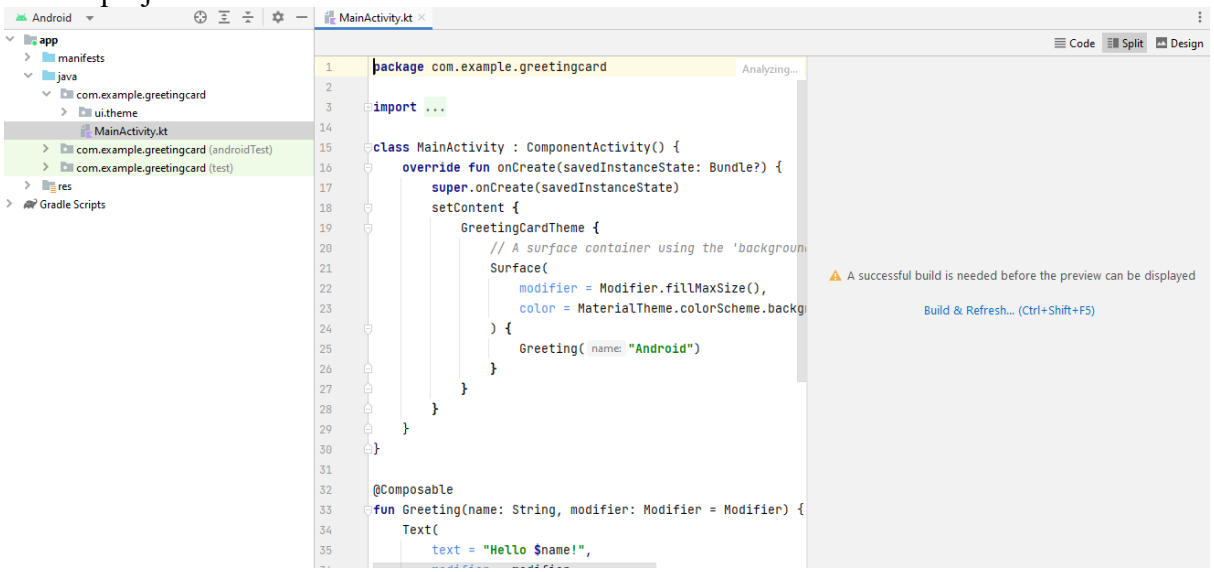
Esse modelo é um projeto simples, que tem uma única tela e mostra o texto "Hello Android!".

- Clique em ***Next***. Você verá alguns campos para configurar o projeto. Eles devem ser preenchidos da seguinte maneira:
  - ***Name*** - é usado para inserir o nome do projeto. Você pode usar “Greeting Card” ou “Cartão de Apresentação” para esse projeto.
  - ***Package name*** - é assim que seus arquivos vão ser organizados na estrutura do arquivo. Deixe esse campo como ele está.

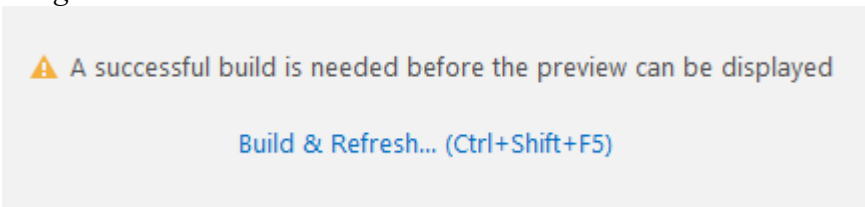
- ***Save location*** - é o local em que todos os arquivos relacionados ao projeto são salvos. Deixe esse campo como ele está.
- ***Language*** - é a linguagem de programação utilizada no seu projeto. A linguagem Kotlin já deve estar selecionada.
- ***Minimum SDK*** - indica a versão mínima do Android em que o seu aplicativo poderá ser executado. Selecione **API 21: Android 5.0 (Lollipop)**.

- Selecione ***Finish***. Isso talvez demore um pouco. Espere todos os arquivos serem carregados.
- Na parte superior direita, clique em ***Split***. Dessa maneira você conseguirá ver o código e o design ao mesmo tempo.

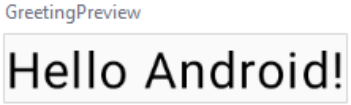
- Perceba que além do código e da aba design, você deve estar vendo uma aba chamada *project* (aba mais a esquerda), que mostra os arquivos e pastas do projeto.



- Na aba *design* é mostrada uma tela em branco com o texto:

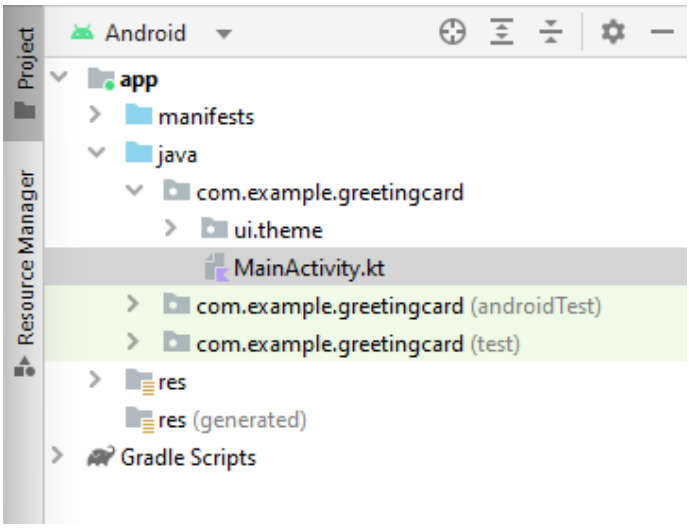


- Clique no botão ***Build & Refresh*** (a criação pode demorar um pouco), em seguida uma caixa de texto com as palavras “Hello Android!” deve aparecer.

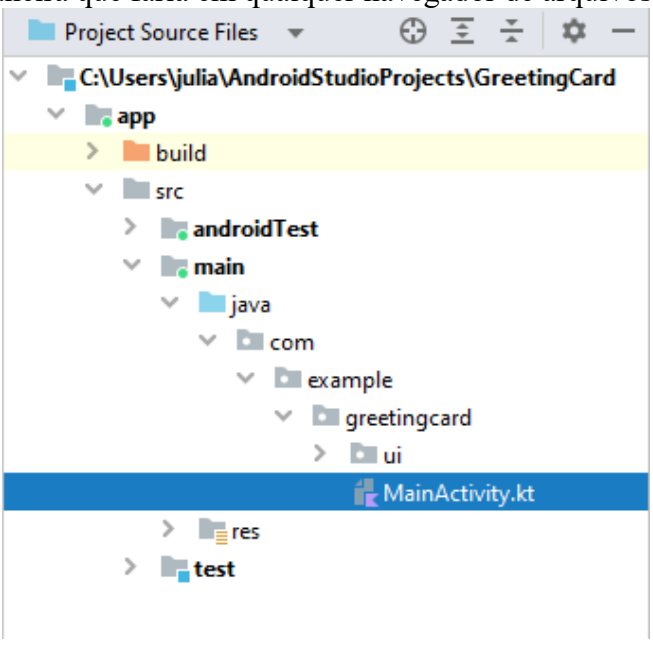


1.2. Conhecendo o Android Studio

No Android Studio, veja a guia *project.* Ela mostra os arquivos e as pastas do seu projeto.



Durante a configuração do projeto, escolhemos o nome do pacote como **com.example.greetingcard.** Um pacote é uma pasta em que o código está localizado.
Selecione ***Project Source Files*** no menu suspenso. Dessa maneira você pode procurar arquivos da mesma maneira que faria em qualquer navegador de arquivos.



Selecione **Android** para retornar à visualização anterior.

1.3. Personalizando sua Carta de Apresentação

Prestando atenção no código do arquivo **MainActivity.kt,** percebemos algumas funções geradas automaticamente pelo modelo.

```
class MainActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
      GreetingCardTheme {
        // A surface container using the 'background' color from the theme
        Surface(
          modifier = Modifier.fillMaxSize(),
          color = MaterialTheme.colorScheme.background
        ) {
          Greeting("Android")
        }
      }
    }
  }
}
```

A função **onCreate()** atua como o ponto de entrada do aplicativo e convoca outras funções para criar a interface do usuário. Em programas escritos em Kotlin, a função **main()** é o local específico no código onde o compilador Kotlin é inicializado. Nos aplicativos Android, a função **onCreate()** desempenha esse papel.
Dentro da função **onCreate()**, a função **setContent()** é utilizada para definir o layout usando funções de composição. Todas as funções que são marcadas com a anotação **@Composable** podem ser chamadas dentro da função **setContent()** ou em outras funções de composição. A anotação informa ao compilador Kotlin que essa função é utilizada pelo *Jetpack Compose* para gerar a interface do usuário.

Uma função de composição recebe uma entrada e gera o que vai ser mostrado na tela.

Observando a função **Greeting()** vemos que ela é uma função de composição e por isso a anotação **@Composable** aparece acima dela.

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
  Text(
    text = "Hello $name!",
    modifier = modifier
  )
}
```

Alguns detalhes:

- Nomes de funções **@Composable** usam letras maiúsculas.
- A anotação **@Composable** é adicionada antes da função.

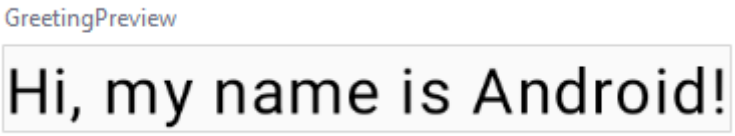
- As funções **@Composable** não podem retornar nada.

No momento, a função **Greeting()** recebe um nome e mostra *"Hello"* para essa pessoa. Para personalizar sua carta de apresentação:

Atualize a função **Greeting()** para apresentar você ao invés de dizer *"Hello"*:

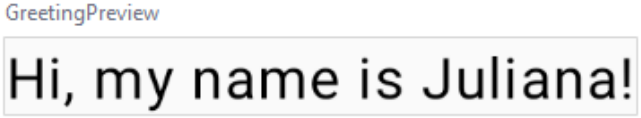
```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
  Text(
     text = "Hi, my name is $name!",
     modifier = modifier
  )
}
```

Pressione o botão de "atualizar" no canto superior esquerdo do painel de design para que seu novo texto apareça:



Agora precisamos personalizar o app para mostrar seu nome, e não "Android". A função **GreetingPreview()** permite ver como o app vai ficar sem que você precise terminar de criá-lo. Para isso, é necessário usar a anotação **@Preview** antes da função.
No nosso caso, a anotação **@Preview** está recebendo o parâmetro "*showBackground*". Se ele for definido como ***true***, um plano de fundo será adicionado à visualização do app. Para atualizar seu nome, basta atualizar a função **GreetingPreview().** Depois disso, atualize e confira seu Cartão.

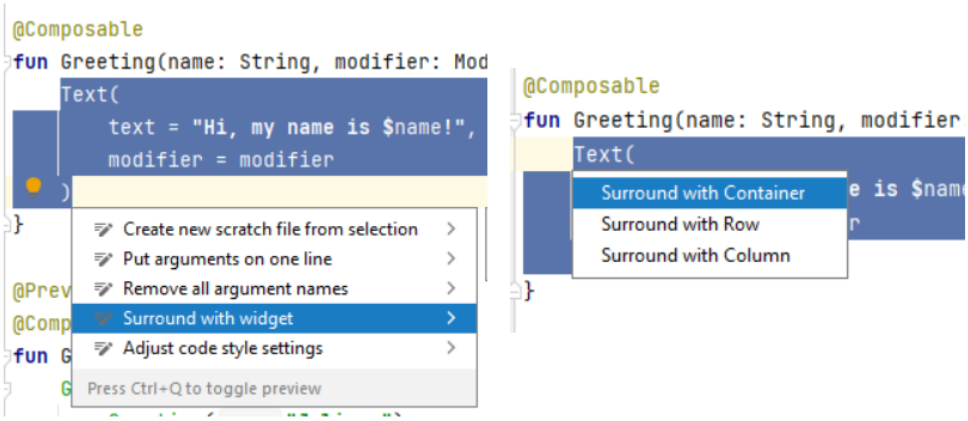
```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
  GreetingCardTheme {
     Greeting("Juliana")
  }
}
```



1.4. Mudando a cor do plano de fundo

Para configurar uma cor de fundo diferente no cartão de apresentação, é necessário envolver o texto com uma ***Surface***, que é um contêiner que representa uma seção da interface em que você pode mudar a aparência do plano de fundo.

- Para fazer isso, destaque a linha do texto, pressione **Alt + Enter** e selecione ***Surround with widget*** e então ***Surround with Container***.



O contêiner padrão é do tipo ***Box***, mas você pode mudá-lo para o tipo que precisar usar:

- Exclua ***Box*** e digite **Surface()**

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
  Surface() {
    Text(
      text = "Hi, my name is $name!",
      modifier = modifier
    )
  }
}
```

- O contêiner ***Surface*** tem um parâmetro ***color***, definido como ***Color***.

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
  Surface(color = Color) {
    Text(
      text = "Hi, my name is $name!",
      modifier = modifier
    )
  }
}
```

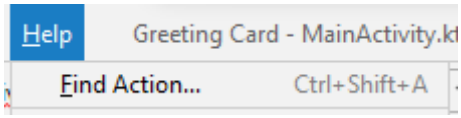
- Ao digitar “Color”, o parâmetro fica vermelho. Para resolver isso, basta ir ao começo do arquivo, abrir a aba de ***import.***

```
import ...
```

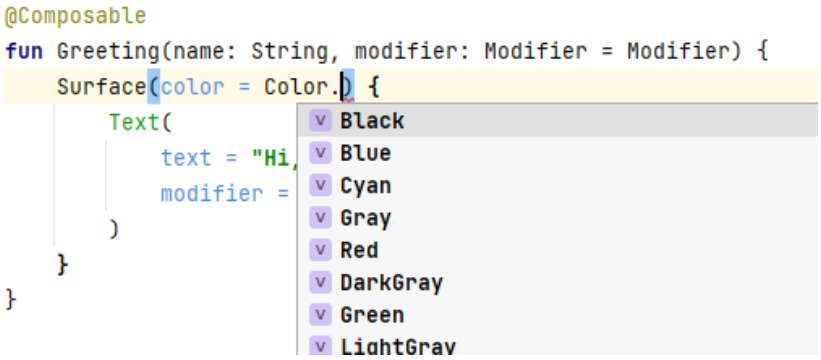
- E adicionar a instrução abaixo no fim da lista.

```
import androidx.compose.ui.graphics.Color
```

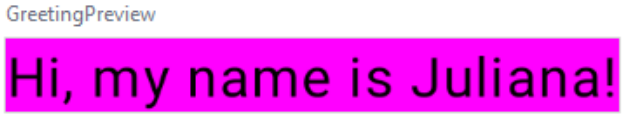
- É recomendado listar as importações em ordem alfabética. Para fazer isso, pressione ***Help*** na barra de ferramentas, clique em ***Find Action…*** e digite ***Optimize Imports***.



- Voltando ao parâmetro ***Color***, percebemos que agora ele está apenas sublinhado em vermelho. Para corrigir esse problema, adicione um ponto depois da palavra ***Color***. Uma lista de cores deve aparecer.



- Escolha uma cor para a superfície, clique em ***Build & Refresh*** novamente e veremos o texto ser cercado pela cor escolhida.



1.5. Adicionando *Padding*

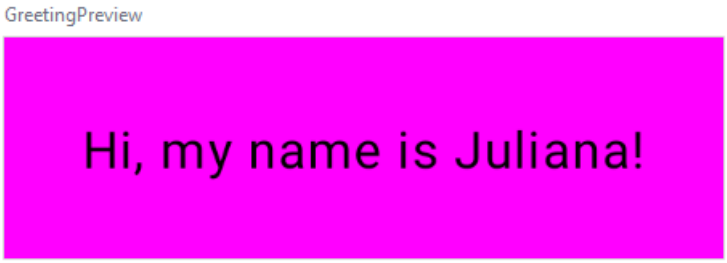
Agora vamos adicionar um espaço (*padding*) ao redor do texto. Para isso podemos usar um ***Modifier***, que é um modificador de elementos. O modificador que usaremos é o ***padding***, que aplica espaço ao redor do elemento. Isso é feito usando a função **Modifier.padding().**

- Adicione as seguintes importações à seção de instruções de importação (não esqueça de otimizar as importações):

```
import androidx.compose.ui.unit.dp
import androidx.compose.foundation.layout.padding
```

- Adicione um *padding* ao texto com tamanho de **24.dp** e clique em ***Build & Refresh***.

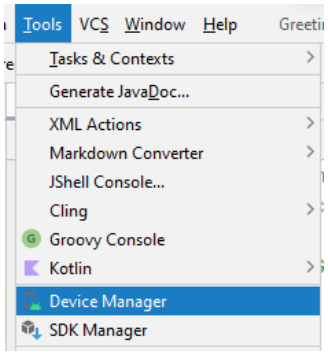
```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
  Surface(color = Color.Magenta) {
    Text(
      text = "Hi, my name is $name!",
      modifier = modifier.padding(24.dp)
    )
  }
}
```



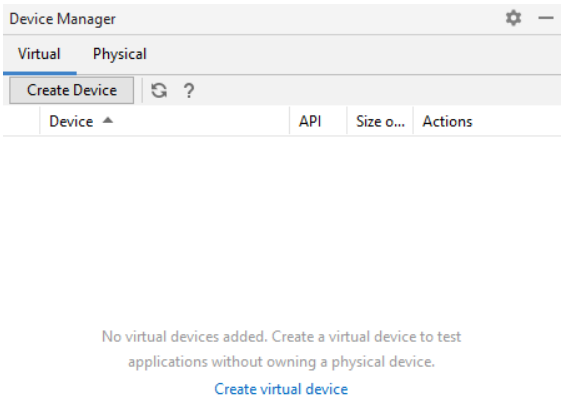
1.6. Executando o aplicativo no Android *Emulator*

Agora vamos criar um Dispositivo Virtual Android (AVD) para executar o nosso aplicativo. Um AVD é um emulador de um dispositivo móvel, que é executado no computador. Pode ser qualquer smartphone, tablet, TV, relógio ou dispositivo Android Auto.

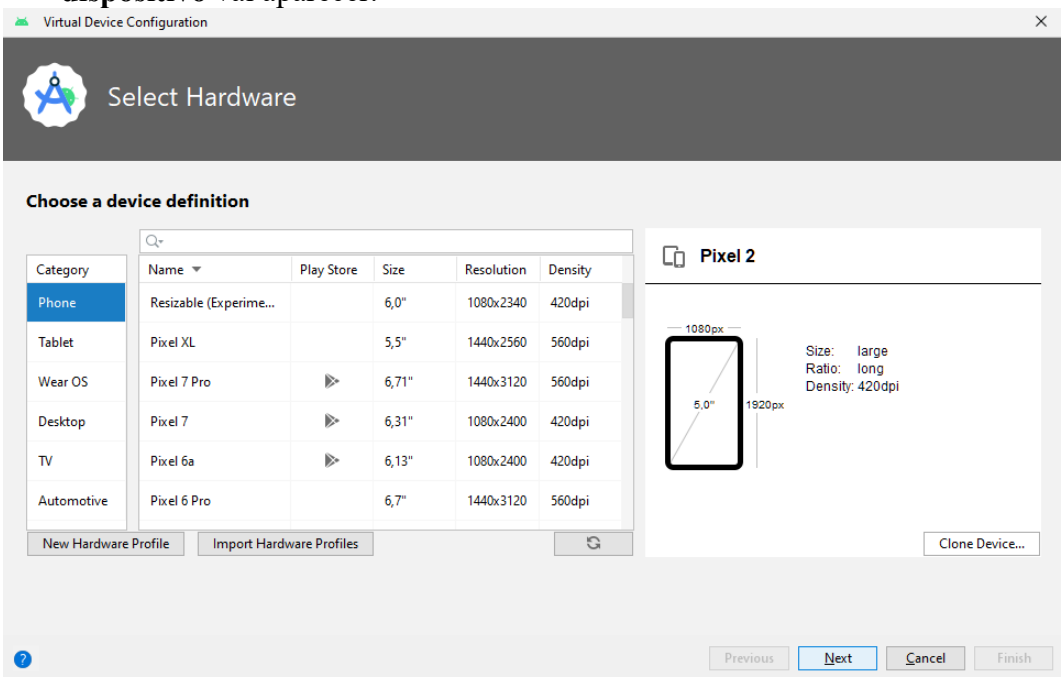
- Para criar um AVD, no Android Studio selecione ***Tools > Device Manager***.



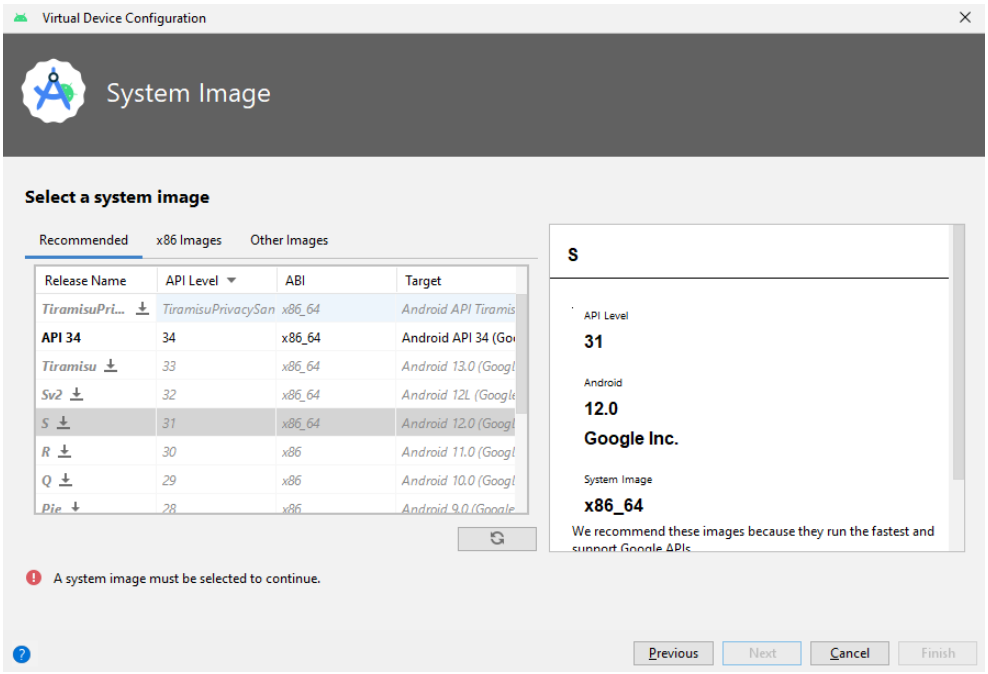
- A caixa de diálogo ***Device Manager*** será aberta.



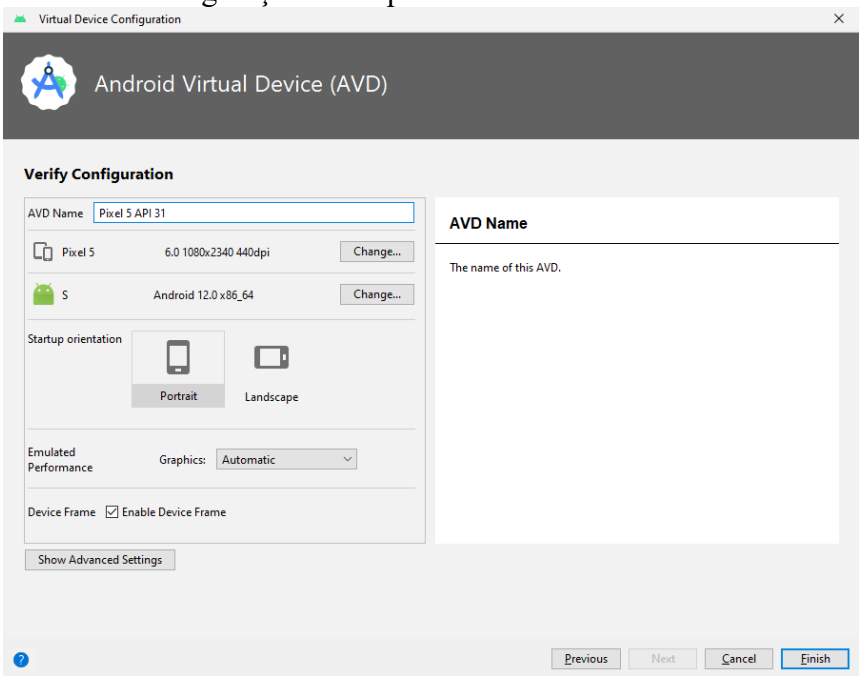
- Clique em ***Create virtual device***. A caixa de diálogo do **Configurador do dispositivo** vai aparecer.



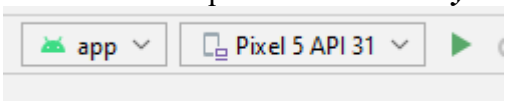
- Selecione ***Phone*** como a categoria e então selecione um smartphone, como o Pixel 5, e clique em ***Next***.
- Agora vamos escolher a versão do Android a ser executada no dispositivo virtual.



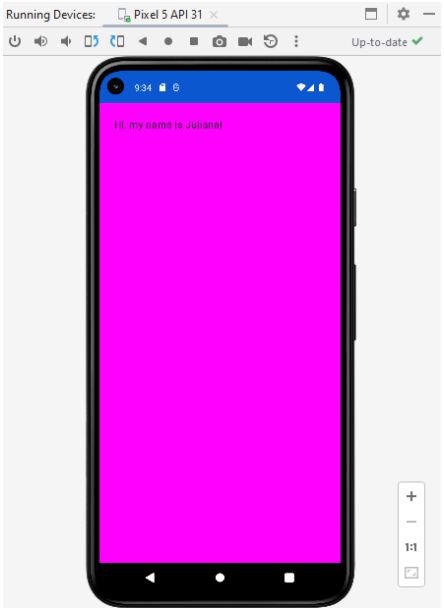
- Caso haja uma indicação de *download* ao lado do S (como mostrado na foto acima), clique para que o download seja efetuado. Isso significa que a imagem não está instalada no seu computador. Aguarde a conclusão do *download*. Isso pode demorar um pouco.
- Após o *download*, selecione S como a versão do Android a ser executada, e clique em ***next***. Isso abrirá uma nova tela, em que você pode escolher detalhes adicionais da configuração do dispositivo.



- No campo **AVD Name**, digite um nome para o seu AVD ou use o padrão. Não mude os outros campos. Em seguida, clique em ***Finish***.
- Essa ação retorna ao painel do ***Device Manager***. Você pode fechá-lo por enquanto.
- Por fim, no menu suspenso na parte superior do Android Studio, selecione o AVD que você criou. E então clique no ícone de ***Play*** ao lado dele.



O dispositivo virtual será inicializado como um dispositivo físico. Pode levar um tempo até que ele inicialize. O emulador será aberto ao lado do editor de código e ele deve exibir o seu aplicativo.

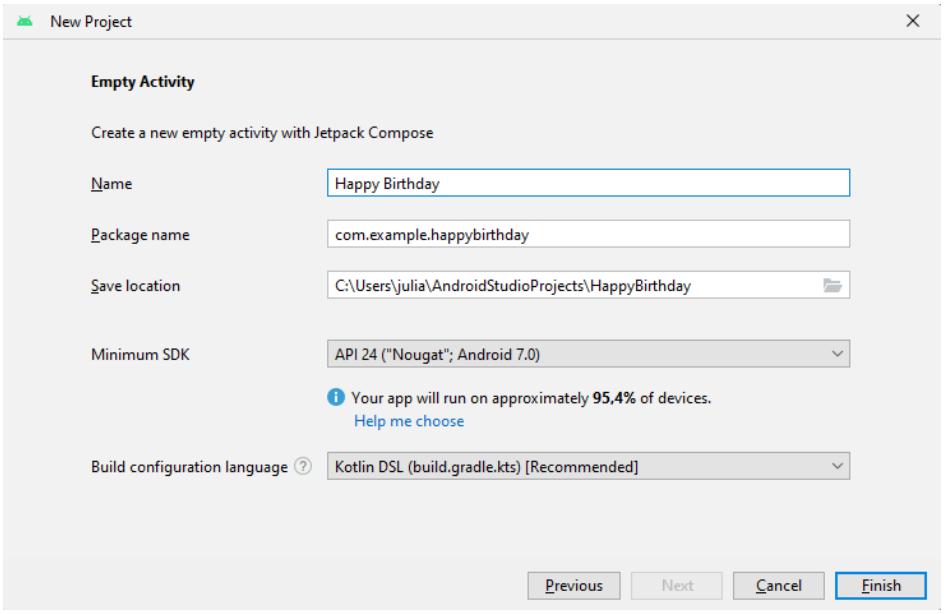


2. Entendendo o Editor de *Layout* do Android Studio

2.1. Criando o novo projeto

Agora vamos criar um novo projeto para fazer um cartão de aniversário.

- Na página inicial do Android Studio, selecione ***New Project -> Empty Activity -> Next.***
- Dê o nome de ***Happy Birthday*** para o seu projeto, selecione o **Android 7.0 (Nougat)** no campo ***Minimum SDK*** e clique em ***Finish***.



- Espere o Android Studio criar o projeto e em seguida clique em ***Run***.
- Você verá a mesma tela que vimos no início do capítulo anterior, inteiramente branca com "*Hello Android*!" escrito.

2.2. O que é uma interface de usuário?

A interface do usuário de um aplicativo é o que é mostrado na tela: texto, imagens, botões e muitos outros tipos de elementos, além de como eles são organizados na tela. É a forma como o *app* mostra as informações e como o usuário interage com ele.
Quase tudo o que você vê na tela do *app* é chamado de componente da interface. Esses componentes podem ser interativos, como um botão clicável ou um campo de entrada editável, ou podem ser imagens decorativas ou informativas.
Neste projeto vamos trabalhar com o elemento da interface que mostra texto, conhecido como elemento ***Text.***

2.3. O que é o *Jetpack Compose*?

O *Jetpack Compose* é um kit de ferramentas moderno que simplifica e acelera o desenvolvimento da interface no Android com menos código, ferramentas poderosas e recursos Kotlin intuitivos. Com o Compose, podemos criar a interface definindo um conjunto de funções, conhecidas como **funções combináveis**, que recebem dados e descrevem elementos da interface. Essas funções combináveis não retornam nada, mas geram o que será mostrado na tela.

Anotações
As anotações são meios de anexar informações extras ao código. Essas informações ajudam o compilador do Jetpack Compose e até outros desenvolvedores a entender o código do app.

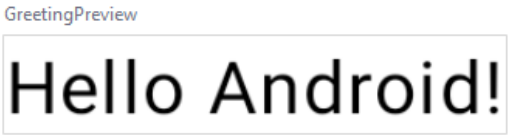
Uma anotação é aplicada utilizando o caractere @ como prefixo. Vários elementos de código, como propriedades, funções e classes, podem ser anotados. Abaixo temos um exemplo de função com anotação:

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {...}
```

Parâmetros

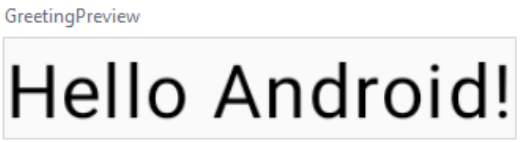
As anotações podem usar parâmetros, que fornecem mais informações para as ferramentas que fazem o processamento delas:

```
@Preview
@Composable
fun GreetingPreview() {
    HappyBirthdayTheme {
        Greeting( name: "Android")
    }
}
```



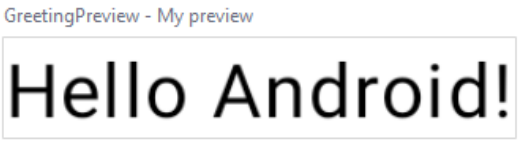
Anotação sem parâmetros

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    HappyBirthdayTheme {
        Greeting( name: "Android")
    }
}
```



Anotação que exibe o plano de fundo da prévia

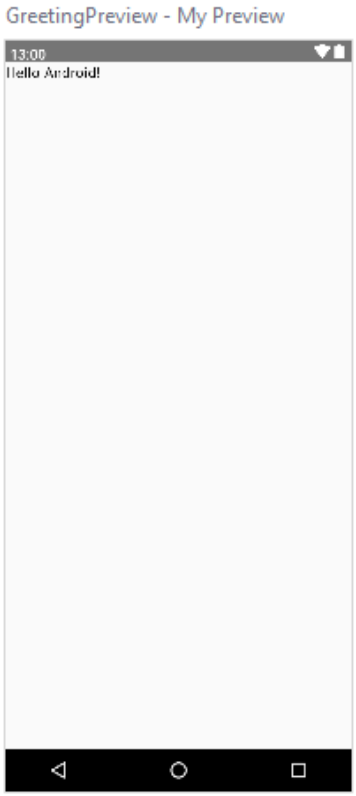
```
@Preview(name = "My preview")
@Composable
fun GreetingPreview() {
    HappyBirthdayTheme {
        Greeting( name: "Android")
    }
}
```



Anotação com o título da prévia

É possível transmitir vários parâmetros para a anotação:

```
@Preview(
    showBackground = true,
    showSystemUi = true,
    name = "My Preview"
)
@Composable
fun GreetingPreview() {
    HappyBirthdayTheme {
        Greeting( name: "Android")
    }
}
```



Anotação com título de visualização e interface do sistema (tela do smartphone).

Funções Combináveis

A função combinável recebe a anotação **@Composable**. Todas as funções desse tipo precisam ter essa anotação. Ela informa ao compilador do Compose que essa função se destina a converter dados em interface.

```
@Composable
fun Greeting(name: String) {
    Text(
        text = "Hello $name!"
    )
}
```

As funções combináveis podem aceitar argumentos. Nesse caso, o elemento da interface aceita uma *String* para que a mensagem use o nome do usuário.

- **Nomes de funções combináveis**

A função de composição que não retorna nada e carrega a anotação @Composable **PRECISA** ser nomeada usando o padrão Pascal Case, que se refere a uma convenção de nomenclatura em que a **primeira letra de cada palavra** em uma palavra composta é maiúscula.

A função Compose:

- PRECISA ser um substantivo: DoneButton()
- NÃO pode ser um verbo ou frase verbal: **Draw**TextField()

- NÃO pode ser uma preposição nominal: TextField**With**Link()
- NÃO pode ser um adjetivo: Bright()
- NÃO pode ser um advérbio: Outside()
- Substantivos PODEM ter adjetivos descritivos como prefixos: RoundIcon()

Além disso, como prática recomendada, as funções devem ser nomeadas para descrever a funcionalidade que elas têm. Portanto:

- Role até a função **GreetingPreview()** e mude o nome da função para **BirthdayCardPreview().**

```
@Preview(showBackground = true)
@Composable
fun BirthdayCardPreview() {
    HappyBirthdayTheme {
        Greeting( name: "Android")
    }
}
```

2.4. Painel *Design* do Android Studio

Anteriormente vimos que é possível visualizar uma prévia da aparência do app no painel ***Design*** do Android Studio. Para que a prévia possa ser mostrada, a função combinável precisa fornecer valores padrão para todos os parâmetros. Por isso, não é recomendado fazer uma prévia direta da função **Greeting().** Aí entra a função **BirthdayCardPreview()** que chama a **Greeting().**

- Vamos conferir uma prévia do seu cartão com seu nome. Para isso, substitua o argumento "Android" na **Greeting()** pelo seu nome.

```
@Preview(showBackground = true)
@Composable
fun BirthdayCardPreview() {
    HappyBirthdayTheme {
        Greeting( name: "Juliana")
    }
}
```

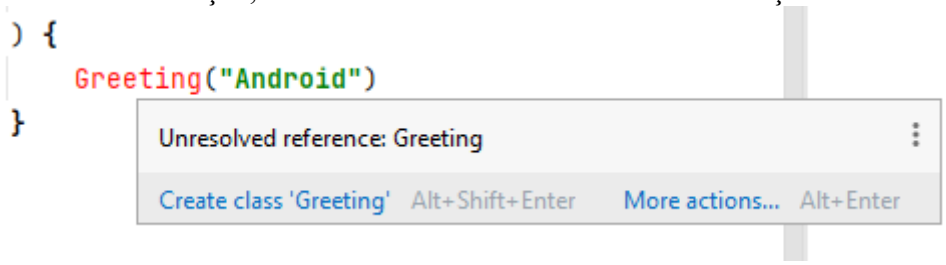
2.5. Adicionando um novo elemento de texto

Agora vamos remover a saudação **Hello $name!** e adicionar uma mensagem de aniversário no lugar.

- No arquivo **MainActivity.kt**, exclua a definição da função **Greeting()**:

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
}
```

- Dentro da função **onCreate()**, observe que a chamada da função **Greeting()** agora está em vermelho. Essa cor indica um erro. Passe o cursor sobre essa chamada de função, e o Android Studio vai mostrar informações sobre o erro.



- Exclua as chamadas da função **Greeting()** tanto na função **onCreate(),** quanto na função **BirthdayCardPreview().** Seu código deverá ficar parecido com este:

```
class MainActivity : ComponentActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContent {
      HappyBirthdayTheme {
        // A surface container using the 'background' color from the theme
        Surface(
          modifier = Modifier.fillMaxSize(),
          color = MaterialTheme.colorScheme.background
        ) {

        }
      }
    }
  }
}

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
  HappyBirthdayTheme {
  }
}
```

- Antes da função **BirthdayCardPreview()**, adicione uma nova função chamada **GreetingText()**. Adicione a anotação **@Composable** antes da função, porque ela será uma função combinável.

```
@Composable
fun GreetingText() {
}
```

- É uma boa prática fazer com que a função combinável aceite um parâmetro **Modifier.** Isso será explicado mais para frente.

```
@Composable
fun GreetingText(modifier: Modifier = Modifier) {
}
```

- Adicione um parâmetro **message** do tipo **String**.

```
@Composable
fun GreetingText(message: String, modifier: Modifier = Modifier) {
}
```

- Dentro da função, adicione uma função combinável **Text** transmitindo a mensagem de texto como um argumento nomeado.

```
@Composable
fun GreetingText(message: String, modifier: Modifier = Modifier) {
  Text(
     text = message
  )
}
```

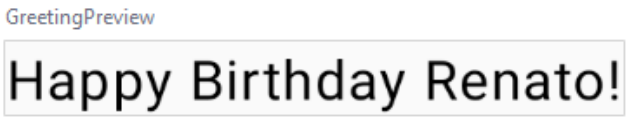
- Agora a função **GreetingText()** mostra texto na interface. Para visualizar essa função, temos que chamá-la dentro da **BirthdayCardPreview().**

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
  HappyBirthdayTheme {
     GreetingText(message = "")
  }
}
```

- Dentro do argumento **String** de **message**, você pode escrever a saudação de aniversário que preferir!

```
@Preview(showBackground = true)
```

```
@Composable
fun GreetingPreview() {
  HappyBirthdayTheme {
     GreetingText(message = "Happy Birthday Renato!")
  }
}
```



2.6. Mudando o tamanho da fonte

Nosso *app* ainda não está com a aparência final. Agora vamos aprender como mudar o tamanho da fonte e a cor do texto. A unidade de medida usada para o tamanho da fonte é de *pixels escalonáveis* (SP, na sigla em inglês).

- No elemento combinável **Text()** na função **GreetingText()**, transmita um argumento **fontSize** à função **Text()** como um segundo argumento nomeado e o defina como um valor de **100.sp**.

```
Text(
  text = message,
  fontSize = 100.sp
)
```

- Android Studio destaca o código **.sp** porque é preciso importar algumas classes ou propriedades para compilar o app. Para isso clique em cima do **.sp** e pressione Alt+Enter.
- Observe a prévia atualizada. O motivo dessa sobreposição é porque precisamos especificar também a altura da linha.

- Atualize o elemento Text para incluir a altura da linha.

```
Text(
  text = message,
  fontSize = 100.sp,
```

```
  lineHeight = 116.sp
)
```

Você pode testar diferentes tamanhos de fonte!

2.7. Adicionando outro elemento de texto

Agora vamos assinar o cartão, com o nome do remetente.

- Adicione um parâmetro ***from*** do tipo ***String*** à função **GreetingText()**.

```
fun GreetingText(message: String, from: String, modifier: Modifier = Modifier)
```

- Após a mensagem de aniversário ***Text***, adicione outro elemento ***Text*** que aceite um argumento ***text*** definido como o valor ***from***.

```
@Composable
fun GreetingText(message: String, from: String, modifier: Modifier = Modifier) {
    Text(
        text = message,
        fontSize = 100.sp,
        lineHeight = 116.sp
    )
    Text(
        text = from
    )
}
```

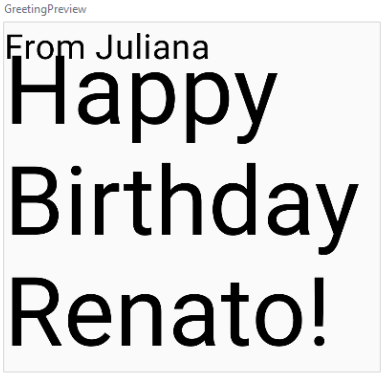
- Adicione um argumento *fontSize* definido com o valor desejado.

```
Text(
  text = from,
  fontSize = 36.sp
)
```

- Na função **BirthdayCardPreview()** adicione outro argumento *String* para assinar o cartão com seu nome.

```
@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    HappyBirthdayTheme {
        GreetingText(message = "Happy Birthday Renato!", from = "From Juliana")
    }
}
```

Como não especificamos a maneira que queremos que os elementos se organizem, o *Compose* os organizou da própria maneira, e eles acabaram se sobrepondo.



2.8. Organizando os elementos do texto

Um componente da interface pode conter um ou mais componentes, e às vezes os termos pai e filho são usados. O contexto é que os elementos pais da interface têm elementos filhos, que por sua vez podem conter outros elementos filhos.

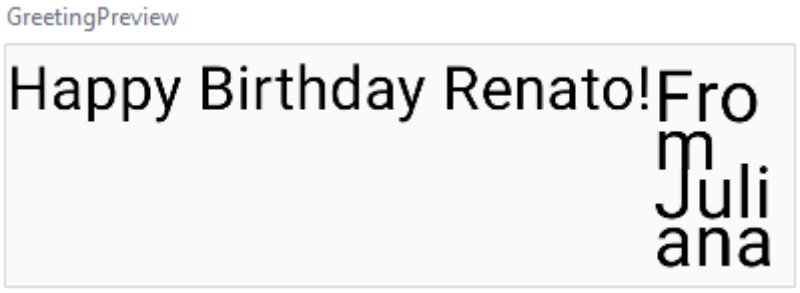
Os três elementos básicos de layout padrão do Compose são ***Column***, ***Row*** e ***Box.*** Eles são funções combináveis que usam conteúdo de composição como argumentos para que você possa colocar itens dentro desses elementos de layout.

Agora vamos organizar os elementos de texto do nosso app em uma linha, para evitar a sobreposição.

- Na função **GreetingText()**, adicione o elemento combinável ***Row*** ao redor dos elementos de texto. Para isso selecione os dois elementos ***Text*** e clique no ícone de lâmpada. Depois, selecione ***Surround with widget > Surround with Row***. A função deverá ficar assim:

```
@Composable
fun GreetingText(message: String, from: String, modifier: Modifier = Modifier) {
    Row { this: RowScope
        Text(
            text = message,
            fontSize = 100.sp,
            lineHeight = 116.sp
        )
        Text(
            text = from,
            fontSize = 36.sp
        )
    }
}
```

- Fazendo isso, o Android Studio importa automaticamente a função *Row*. Mude temporariamente o tamanho da fonte da mensagem de aniversário para 30.sp e observe a prévia.

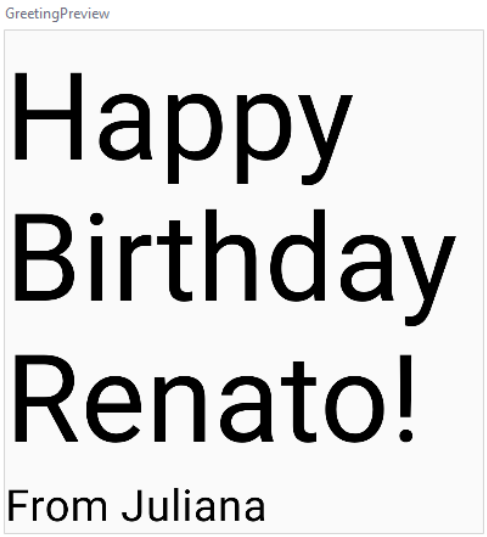


As duas frases não estão mais se sobrepondo, no entanto ainda não temos espaço suficiente para a assinatura. Para resolver isso, vamos organizar os elementos do texto em uma coluna.

- Para fazer isso, basta realizar o mesmo passo a passo anterior, mas utilizar ***column*** ao invés de ***row***. Tente você mesmo! Seu código deverá ficar assim.

```
@Composable
fun GreetingText(message: String, from: String, modifier: Modifier = Modifier) {
    Column { this: ColumnScope
        Text(
            text = message,
            fontSize = 100.sp,
            lineHeight = 116.sp
        )
        Text(
            text = from,
            fontSize = 36.sp
        )
    }
}
```

Com isso, temos o resultado:



Obs: É recomendável transmitir os atributos de modificador junto ao modificador do elemento combinável pai.

2.9. Adicionando a saudação ao *APP*

Quando estiver contente com a prévia, é hora de adicionar o elemento combinável ao app no seu dispositivo ou emulador.

- Na função **onCreate()**, chame a função **GreetingText()** do bloco **Surface**, e transmita a mensagem de aniversário. Sua função deve ficar parecida com isso:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            HappyBirthdayTheme {
                // A surface container using the 'background' color from the theme
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    GreetingText(message = "Happy Birthday Renato!", from = "From Juliana" )
                }
            }
        }
    }
}
```

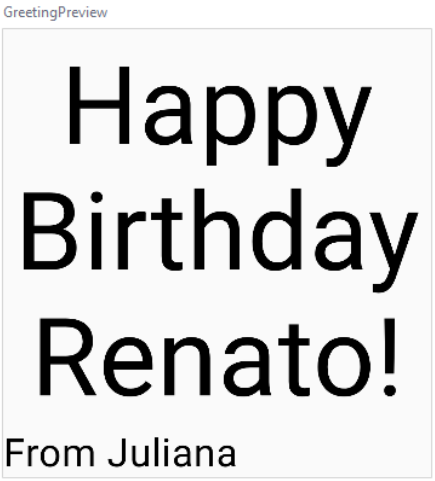
- Execute o *APP* no emulador.

- Se você quiser, podemos centralizar a saudação. Para isso, adicione um parâmetro chamado **verticalArrangement** e defina-o como **Arrangement.Center**.

```
@Composable
fun GreetingText(message: String, from: String, modifier: Modifier = Modifier) {
  Column(
     verticalArrangement = Arrangement.Center
   ){
    //...
  }
}
```

- Em seguida, centralize o texto da saudação, utilizando ***textAlign***.

```
Text(
  text = message,
  fontSize = 100.sp,
  lineHeight = 116.sp,
  textAlign = TextAlign.Center
)
```



Vemos que somente a mensagem de parabéns está centralizada e que a assinatura está muito próxima à borda. Agora vamos alinhar a assinatura à direita e adicionar um *padding* à ela, para que ela não fique tão grudada.

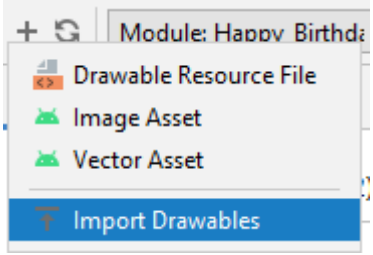
```
Text(
  text = from,
  fontSize = 36.sp,
  modifier = Modifier
    .padding(16.dp)
    .align(alignment = Alignment.End)
)
```

Lembre-se que é recomendável transmitir os atributos de modificador junto ao modificador do elemento combinável pai.

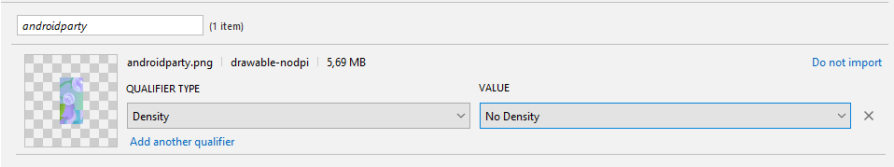
3. Adicionando imagens ao *APP* Android

3.1. Configurando o app

- Baixe uma imagem (arquivo .png) para ser o fundo do seu cartão de aniversário.
- No Android Studio, clique em *View* > *Tool Windows* > *Resource Manager*.
- Clique em + (*Add resources to the module*) > *Import Drawables*.

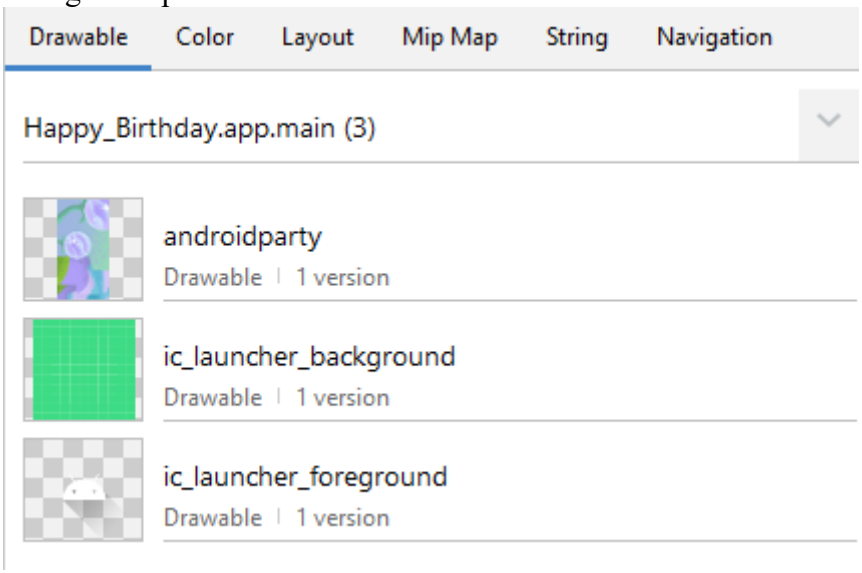


- No navegador de arquivos, selecione o arquivo de imagem que você transferiu por download e clique em Open. Essa ação abre a caixa de diálogo *Import Drawables*.
- Na lista ***QUALIFIER TYPE*** selecione ***Density***, e na lista ***VALUE*** selecione ***NO DENSITY***.

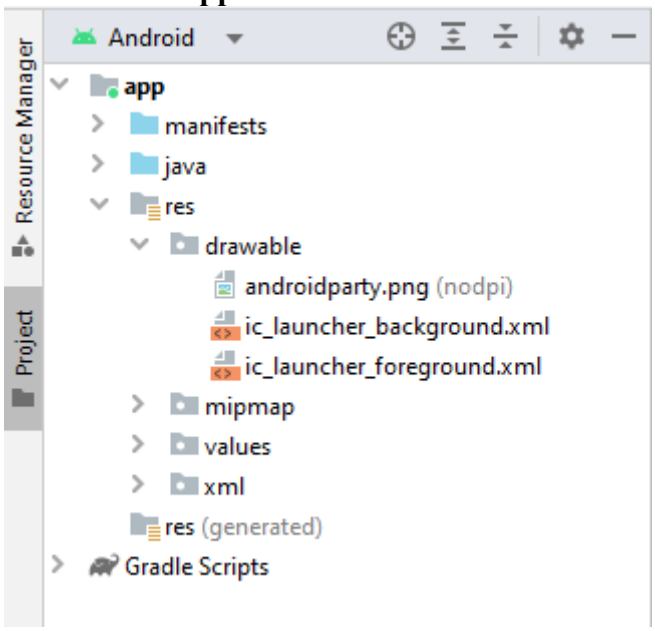


- Clique em Next e em seguida clique em Import.

Caso a imagem seja importada corretamente, ela será adicionada pelo Android Studio à lista na guia **Drawable**. Essa lista inclui todos os ícones e imagens do app, então você já pode usar a imagem importada.



- Na lateral esquerda do Android Studio, volte à aba **Project**, e confirme que a imagem está no caminho **app > res > drawable.**



3.2. Incluindo uma função combinável para adicionar uma imagem

Para que uma imagem seja mostrada no app, precisamos informar o local de exibição. Do mesmo modo que usamos um elemento combinável **Text** para mostrar texto, é possível usar um elemento **Image** para mostrar uma imagem.

- No arquivo **MainActivity.kt**, adicione uma função combinável **GreetingImage()** após a função **GreetingText()**.
- Transmita dois parâmetros String à ela: o message e o from. Adicione também o parâmetro modifier. Ele informa para um elemento da interface como esses elementos serão dispostos, exibidos ou se comportarão no layout pai.

```
@Composable
fun GreetingImage(message: String, from: String, modifier:
Modifier = Modifier){

}
```

- Na função **GreetingImage()**, declare uma propriedade **val** e a nomeie como **image**.
- Faça uma chamada para a função **painterResource()** transmitindo o recurso **androidparty**. Para acessar os recursos é necessário usar as IDs do projeto. No nosso caso, o resurso androidparty seria acessado pela ID **R.drawable.graphic .** R é a classe gerada automaticamente pelo Android que contém os IDs de todos os recursos do projeto. **Drawable** é o subdiretório dentro da pasta **res** em que a imagem está contida. Atribua o valor retornado à variável **image**.

```
@Composable
fun GreetingImage(message: String, from: String, modifier:
Modifier = Modifier){
   val image = painterResource(R.drawable.androidparty)
```

```
}
```

- Após a chamada para a função **painterResource()**, adicione um elemento combinável **Image** e, em seguida, transmita a image para o painter como um argumento nomeado.

```
Image(
    painter = image
)
```

- Adicione as importações necessárias utilizando **Alt + Enter** onde necessário.

Você pode adicionar elementos que melhorem a acessibilidade do seu aplicativo. Um deles é o **contentDescription**, que descreve um elemento para que o seu aplicativo seja melhor utilizado com o *TalkBack*. Como nossa imagem só foi adicionada para fins decorativos, uma descrição da imagem dificultaria o uso com o TalkBack neste caso específico. Então podemos definir o argumento **contentDescription** como **null**, para que o *TalkBack* ignore esse elemento.

```
Image(
    painter = image,
    contentDescription = null
)
```

Para visualizar nosso elemento Image, basta substituir a função **GreetingText()** pela função **GreetingImage()** dentro da **BirthdayCardPreview()**. Seu código deverá ficar parecido com esse:

```
@Preview(showBackground = true)
@Composable
fun BirthdayCardPreview() {
    HappyBirthdayTheme {
        GreetingImage("Happy Birthday Renato!", "From Juliana")
    }
}
```

Com isso, veremos a tela abaixo. Não é mais possível ver o texto, pois a nova função tem apenas um elemento **Image**, mas não tem nenhum **Text**.

3.3. Adicionando o layout de caixa

Com o layout **Box** podemos empilhar elementos uns sobre os outros.

- Na função **GreetingImage()**, adicione um elemento **Box** ao redor do elemento **Image**. O código ficará como abaixo:

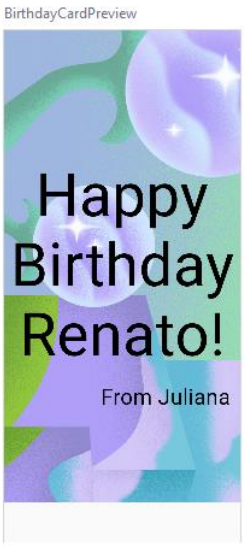
```
@Composable
fun GreetingImage(message: String, from: String, modifier:
Modifier = Modifier){
    val image = painterResource(R.drawable.androidparty)
    Box{
        Image(
            painter = image,
            contentDescription = null
        )
    }
}
```

- Abaixo da função Image, chame a função **GreetingText()**, e transmita a ela a mensagem de aniversário, a assinatura e o modificador, conforme mostrado abaixo:

```
@Composable
fun GreetingImage(message: String, from: String, modifier:
Modifier = Modifier) {
    val image = painterResource(R.drawable.androidparty)
    //Step 3 create a box to overlap image and texts
    Box {
        Image(
            painter = image,
            contentDescription = null
```

```
        )
        GreetingText(
            message = message,
            from = from,
            modifier = Modifier
                .fillMaxSize()
                .padding(8.dp)
        )
    }
}
```

Agora podemos ver as palavras no cartão:



Para que as mudanças acima sejam aplicadas no emulador ou em um dispositivo, na função **onCreate()**, substitua a chamada de função **GreetingText()** por **GreetingImage()**. A imagem tem a mesma largura da tela, mas está fixada na parte de cima. Há um espaço em branco na parte de baixo da tela, o que não fica muito interessante. Em seguida, vamos aprender a redimensionar imagens.

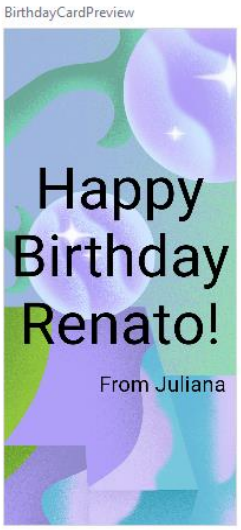
3.4. Dimensionando a imagem e mudando a opacidade

Para redimensionar a imagem, podemos utilizar o parâmetro **ContentScale.** Há vários tipos de ContentScale disponíveis, mas nós usaremos o **ContentScale.Crop,** que não muda a proporção da imagem.

- Adicione um argumento nomeado **ContentScale** à imagem e importe a propriedade necessária.

```
Image(
    painter = image,
    contentDescription = null,
    contentScale = ContentScale.Crop
)
```

Agora a imagem preenche toda a tela.



Para melhorar o contraste do app, vamos diminuir a opacidade do plano de fundo.

- Adicione o parâmetro alpha à imagem e defina-o como 0.5F

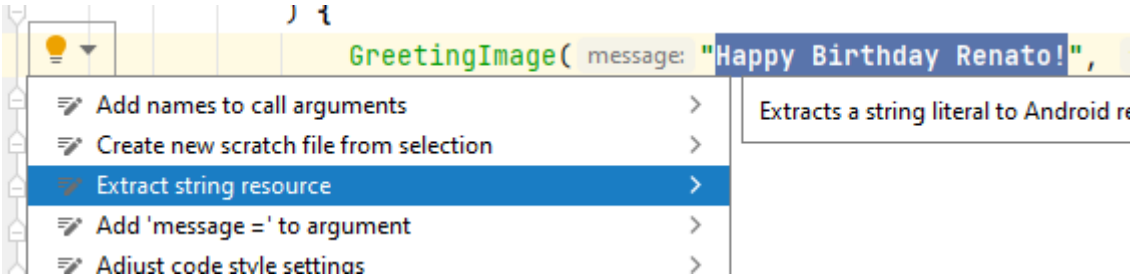
```
Image(
    painter = image,
    contentDescription = null,
    contentScale = ContentScale.Crop,
    alpha = 0.5F
)
```

Perceba a diferença na opacidade.

3.5. Boas práticas de código

É importante lembrar que nossos apps podem ser traduzidos para outros idiomas em algum momento. Para facilitar essa tradução, podemos extrair as Strings que estão escritas diretamente no nosso código para um arquivo de recursos. Dessa maneira, em vez de fixar strings no código, colocaremos em um arquivo, nomearemos os recursos de string e utilizaremos os nomes sempre que quisermos usar as strings.

- No arquivo **MainActivity.kt**, role até a função **onCreate()**. Selecione a mensagem de aniversário, string **Happy Birthday Renato!** sem aspas.
- Clique na lâmpada no lado esquerdo da tela, e selecione **Extract string resource**.



- O Android Studio vai abrir a caixa de diálogo **Extract Resource**. Nela, mude o resource name para **happy_birthday_text**.
- Perceba as mudanças no código.

```
HappyBirthdayTheme {
    Surface(
        modifier = Modifier.fillMaxSize(),
        color = MaterialTheme.colorScheme.background
    ) {
        GreetingImage(stringResource(R.string.happy_birthday_text),
"From Juliana")
```

```
    }
}
```

Obs: algumas versões do Android Studio substituem a string fixada no código pela função **getString()**. Nesses casos, mude manualmente a função para **stringResource()**.

- Repita essas etapas para a String da assinatura, mas dessa vez utilize **signature_text** como Resource name
- Você pode ver o **arquivo de recursos** no caminho **app > res > values > strings.xml**

3.6. Boas práticas de código

Organize ou alinhe o elemento combinável de texto de assinatura para que ele fique centralizado na tela.

Dica: o Compose oferece o modificador **align** para posicionar um elemento combinável filho individualmente, desafiando as regras de layout de alinhamento aplicadas pelo layout pai. Encadeie o argumento .align(alignment = Alignment.CenterHorizontally) com o texto combinável Modifier.
Caso a imagem seja importada corretamente, ela será adicionada pelo Android Studio à lista na guia Drawable. Essa lista inclui todos os ícones e imagens do app, então você já pode usar a imagem importada.

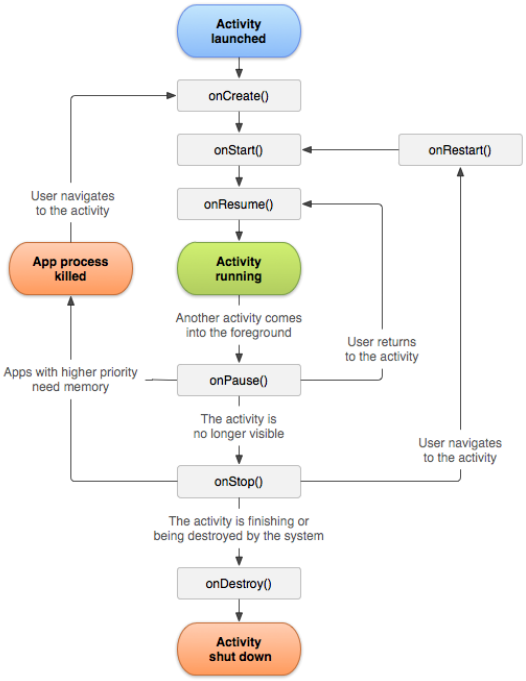
4. Estágios do ciclo de vida da atividade

O ciclo de vida de uma atividade é o nome dado à transição de estados que uma atividade passa. Uma atividade pode mostrar várias telas e alternar entre elas, conforme necessário. O ciclo de vida da atividade se estende desde a criação até a destruição da atividade, quando o sistema recupera os recursos dela. À medida que o usuário navega para dentro e para fora de uma atividade, cada atividade transita entre diferentes estados no ciclo de vida.
Nessa etapa, vamos utilizar um aplicativo inicial chamado “Dessert Clicker”. Nele, sempre que o usuário tocar em uma sobremesa na tela, o app a "compra" para o usuário. O app atualiza valores no layout para o número de sobremesas que foram "compradas" e para o custo total das sobremesas "compradas".

4.1. Examinar o método onCreate() e adicionar registros

- Baixe o código em: https://github.com/google-developer-training/basic-android-kotlin-compose-training-dessert-clicker/tree/starter
- No Android Studio, abra o projeto basic-android-kotlin-compose-training-dessert-clicker.
- Execute o app **DessertClicker** e toque várias vezes na foto de uma sobremesa. Observe como o valor de Desserts sold e o valor total em dólares mudam.

Para descobrir o que está acontecendo com o ciclo de vida do Android, é útil saber quando os vários métodos de ciclo de vida são chamados. Uma maneira simples de determinar essas informações é usar a funcionalidade de geração de registros do Android, que permite que você escreva mensagens curtas em um console enquanto o app está em execução.



- Adicione a constante a seguir no nível superior do **MainActivity.kt**, acima da declaração de classe **class MainActivity**.

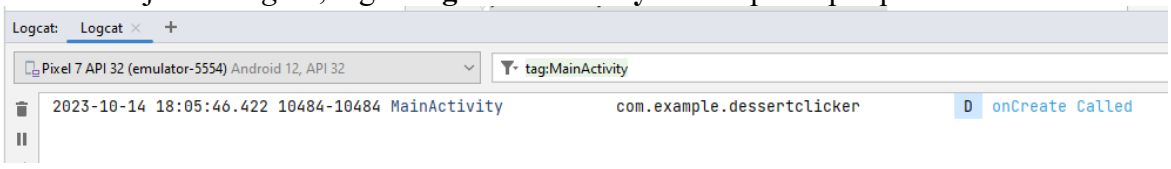
```
private const val TAG = "MainActivity"
```

- No método **onCreate()**, logo após a chamada para **super.onCreate()**, adicione a seguinte linha, e importe a classe necessária.

```
Log.d(TAG, "onCreate Called")
```

A classe *Log* escreve mensagens no *Logcat*, que é o console usado para registrar mensagens do Android sobre seu app.
A mensagem de registro chamada msg (o segundo parâmetro) é uma *string* curta. Neste caso, "**onCreate Called**".

- Compile e execute o app *Dessert Clicker*. Você não vai notar diferenças de comportamento no app quando tocar na sobremesa. No Android Studio, na parte de baixo da tela, clique na guia *Logcat*.
- Na janela *Logcat*, digite **tag:MainActivity** no campo de pesquisa.



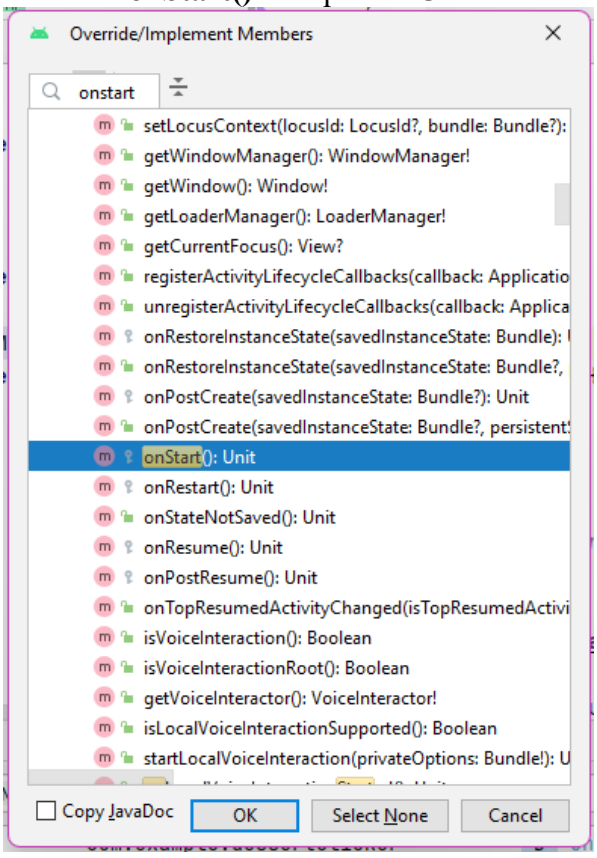
Com isso estamos filtrando as mensagens do **Logcat**. Como utilizamos a MainActivity como tag de registro no código, é possível usá-la para filtrar o registro. Como essa mensagem aparece no registro, você sabe que o método **onCreate()** foi executado.

4.2. Implementar o método onStart()

O método **onStart()** é chamado logo depois de **onCreate()**. Depois que **onStart()** é executado, sua atividade fica visível na tela. Ao contrário de **onCreate()**, que é chamado apenas uma vez para inicializar a atividade, **onStart()** pode ser chamado pelo sistema várias vezes durante o ciclo de vida da atividade.

Todo **onStart()** é pareado com um método do ciclo de vida **onStop()** correspondente. Se o usuário inicia o app e retorna à tela inicial do dispositivo, a atividade é interrompida e não fica mais visível na tela.

- No Android Studio, com MainActivity.kt aberto e o cursor na classe MainActivity, selecione **Code > Override Methods…**. Uma caixa de diálogo vai aparecer com uma longa lista de todos os métodos que você pode substituir nessa classe. Selecione **onStart()** e clique em **OK**.



- No código criado adicione a linha com a TAG para o método onStart(). Seu código deve ficar assim:

```
override fun onStart() {
   super.onStart()
   Log.d(TAG, "onStart Called")
```

```
}
```

- Compile e execute o app *Dessert Clicker* e abra o painel do *Logcat* novamente com a **tag:MainActivity** no campo de pesquisa.

Agora podemos ver que os métodos **onCreate()** e **onStart()** foram chamados um após o outro. Tente sair da tela do aplicativo e depois abrí-la novamente. Você perceberá que o **onStart()** é registrado uma segunda vez no *Logcat*, enquanto o **onCreate()** não é chamado de novo.

Por fim, substitua os métodos restantes do ciclo de vida na **MainActivity** e adicione *log statements* para cada um, como mostrado neste código:

```
override fun onResume() {
    super.onResume()
    Log.d(TAG, "onResume Called")
}

override fun onRestart() {
    super.onRestart()
    Log.d(TAG, "onRestart Called")
}

override fun onPause() {
    super.onPause()
    Log.d(TAG, "onPause Called")
}

override fun onStop() {
    super.onStop()
    Log.d(TAG, "onStop Called")
}

override fun onDestroy() {
    super.onDestroy()
    Log.d(TAG, "onDestroy Called")
}
```

Agora que configuramos o *Dessert Clicker* para a geração de registros, já podemos começar a usar o app e explorar como os *callbacks* do ciclo de vida são acionados. Dessa maneira, podemos encontrar falhas no nosso aplicativo. Vamos começar com um caso de uso básico: fechar e abrir o aplicativo.

- Compile e execute o app *Dessert Clicker*, se ele ainda não estiver em execução. Os *callbacks* **onCreate(), onStart()** e **onResume()** são chamados quando a atividade é inicializada pela primeira vez.
- Toque no *cupcake* algumas vezes e em seguida volte para a tela inicial do celular.
- No *Logcat*, **onPause()** e **onStop()** são chamados nessa ordem.

```
2023-10-15 22:00:05.479 25841-25841 MainActivity          com.example.dessertclicker    D  onCreate Called
2023-10-15 22:00:05.541 25841-25841 MainActivity          com.example.dessertclicker    D  onStart Called
2023-10-15 22:00:05.545 25841-25841 MainActivity          com.example.dessertclicker    D  onResume Called
2023-10-15 22:00:28.666 25841-25841 MainActivity          com.example.dessertclicker    D  onPause Called
2023-10-15 22:00:29.094 25841-25841 MainActivity          com.example.dessertclicker    D  onStop Called
```

Nesse caso, o uso do botão Voltar faz com que a atividade e o app sejam removidos da tela e movidos para a parte de trás da pilha de atividades. Isso coloca o aplicativo em **segundo plano.** Quando **onPause()** é chamado, o app não está mais em foco. Depois de **onStop()**, o app não fica mais visível na tela.

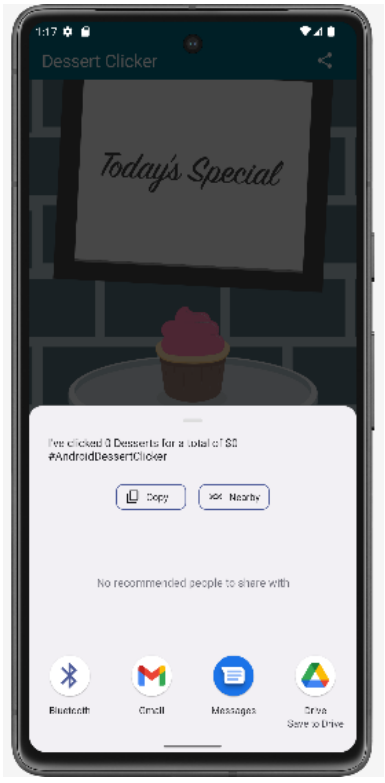
- Use a tela "Recentes" para retornar ao app. No Logcat, a atividade é reiniciada por **onRestart()** e **onStart()** e depois é retomada por **onResume().**

```
MainActivity    com.example.dessertclicker    D  onPause Called
MainActivity    com.example.dessertclicker    D  onStop Called
MainActivity    com.example.dessertclicker    D  onRestart Called
MainActivity    com.example.dessertclicker    D  onStart Called
MainActivity    com.example.dessertclicker    D  onResume Called
```

Quando a atividade retorna para o primeiro plano, o método **onCreate()** não é chamado novamente. O objeto da atividade não foi destruído, por isso não precisa ser criado novamente.

A diferença entre foco e visibilidade é importante. É possível que uma atividade fique parcialmente visível na tela, mas não tenha o foco do usuário.

- Com o app *Dessert Clicker* em execução, clique no botão **Compartilhar** na parte superior direita da tela.



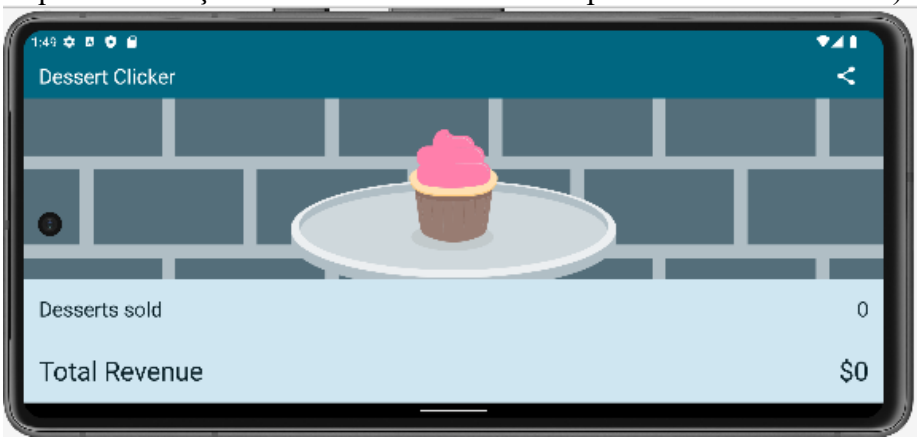
A atividade de compartilhamento vai aparecer na metade de baixo da tela, mas a atividade do app ainda vai estar visível na metade de cima. Analisando o *Logcat*, vemos que apenas o **onPause()** foi chamado. Neste caso de uso, **onStop()** não é chamado, porque a atividade ainda está parcialmente visível. Mas ela não tem o foco do usuário, e não é possível interagir com ela. A atividade de "compartilhamento" que está em primeiro plano tem o foco do usuário.

- Clique fora da caixa de diálogo de compartilhamento para retornar ao app. Observe que o **onResume()** é chamado.

4.3. Explorando mudanças de configuração

Uma mudança de configuração ocorre quando o estado do dispositivo muda de forma tão radical que a maneira mais fácil do sistema resolver a mudança é encerrar completamente e recriar a atividade.

- Com o aplicativo aberto e sendo executado, clique algumas vezes no doce.
- Em seguida, gire o dispositivo para o modo paisagem (certifique-se que o bloqueio de rotação de tela no emulador ou dispositivo está desativado).



Analise o *Logcat* e entenda que, quando a atividade é encerrada, ele chama **onPause(), onStop()** e **onDestroy()**, nessa ordem. Perceba que quando o dispositivo é girado e a atividade é encerrada e recriada, ela é reiniciada com valores padrão. Ou seja, a imagem da sobremesa, o número de sobremesas vendidas e a receita são redefinidos como zero, quando na realidade, não queríamos que isso acontecesse. Vamos ver como consertar esse problema.

A IU do app é criada inicialmente pela execução de funções de composição. Essas funções têm um **ciclo de vida próprio** que é independente do ciclo de vida da atividade. O ciclo de vida delas é composto pelos seguintes eventos: entrar na composição, recompor 0 ou mais vezes e, depois, sair da composição. Para que o *Compose* acompanhe e acione uma recomposição, ele precisa saber quando o estado mudou. Para indicar ao *Compose* que ele precisa rastrear o estado de um objeto, o objeto precisa ser do tipo **State** ou **MutableState**.

Assim, criamos a variável mutável **revenue**, declarando-a usando **mutableStateOf**. 0 é o valor padrão inicial (não é necessário adicionar esse código no seu projeto).

```
var revenue = mutableStateOf(0)
```

Esse código, porém, apenas aciona uma recomposição caso o valor da receita mude. Se o elemento combinável for executado novamente, ele vai reinicializar o valor da receita para o valor padrão inicial de 0. Para instruir o Compose a reter e reutilizar o valor durante as recomposições, é necessário declarar o valor com a API **remembre** (esse código já está no seu projeto).

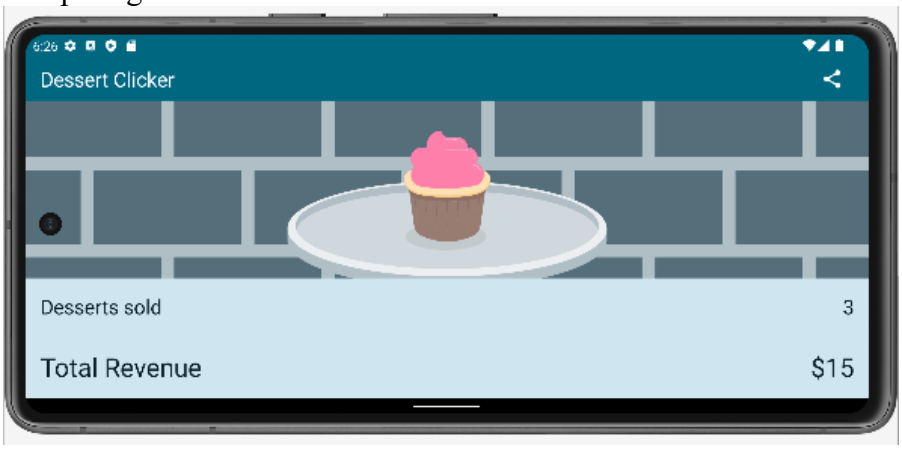
```
var revenue by remember {mutableStateOf(0)}
```

Dessa maneira, se o valor de revenue mudar, o Compose vai programar a recomposição de todas as funções de composição que leem esse valor.
Embora o Compose lembre o estado da receita durante as recomposições, ele ainda não retém esse estado durante uma mudança de configuração. Para que o Compose retenha o estado durante uma mudança de configuração, use **rememberSaveable**.

- No **MainActivity**, atualize o grupo de cinco variáveis que atualmente usam **remember** para **rememberSaveable**.

```
var revenue by rememberSaveable {mutableStateOf(0)}
```

- Compile e execute seu código.
- Clique no cupcake algumas vezes e observe que as sobremesas vendidas e a receita total não são zero. Em seguida, gire o dispositivo ou o emulador para o modo paisagem.



Observe que, depois que a atividade é destruída e recriada, a imagem da sobremesa, as sobremesas vendidas e a receita total são restauradas para os valores anteriores.

5. Componentes de Arquitetura

Agora vamos aprender a criar um app com eficiência e preservar os dados dele durante mudanças de configuração, usando as diretrizes da biblioteca Android Jetpack, do *ViewModel* e da arquitetura de apps Android.
A arquitetura de apps é um conjunto de regras de design para um app. Assim como a planta de uma casa, a arquitetura fornece a estrutura. Uma boa arquitetura pode deixar seu código otimizado, flexível, escalonável, testável e sustentável por anos.
Nesta etapa, vamos utilizar o aplicativo "*Unscramble*", que é um jogo de palavras embaralhadas. O app exibe uma palavra embaralhada, e o jogador precisa adivinhá-la usando todas as letras mostradas. O jogador marca pontos se a palavra está correta. Caso contrário, o jogador pode tentar adivinhar a palavra quantas vezes quiser. O app também

tem a opção de pular a palavra atual. No canto superior direito, ele mostra a contagem de palavras, que é o número de palavras embaralhadas jogadas na sessão atual. Cada partida tem 10 palavras embaralhadas.

- Baixe o projeto no link abaixo, na ramificação ***starter*** https://github.com/google-developer-training/basic-android-kotlin-compose-training-unscramble.git
- Execute o projeto e toque nos botões do app para testá-lo.

Você vai notar bugs no app. A palavra embaralhada não é mostrada, mas está fixada no código como "*scrambleun*", e nada acontece quando você toca nos botões.

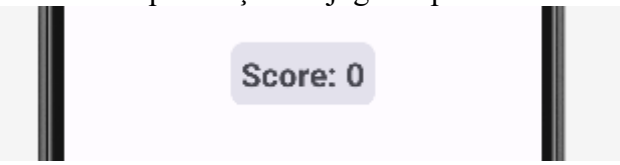
5.1. Código Inicial

Neste projeto, temos três arquivos **.kt** principais:

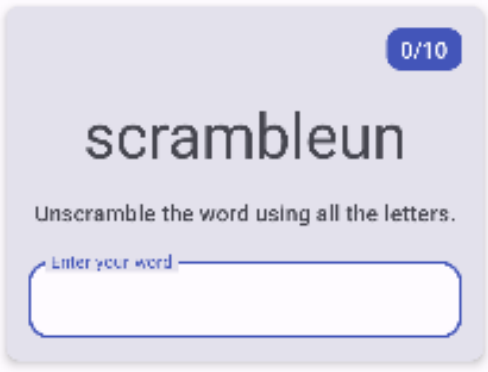
- **WordsData.kt** - contém uma lista das palavras usadas no jogo, constantes para o número máximo de palavras por partida e o número de pontos que o jogador marca para cada palavra correta.
- **MainActivity.kt** - contém principalmente o código gerado pelo modelo. O elemento combinável GameScreen é mostrado no bloco setContent{}.
- **GameScreen.kt** - define todos os elementos combináveis de interface

Algumas funções criadas no nosso projeto são:

- **GameStatus** - mostra a pontuação do jogo na parte de baixo da tela

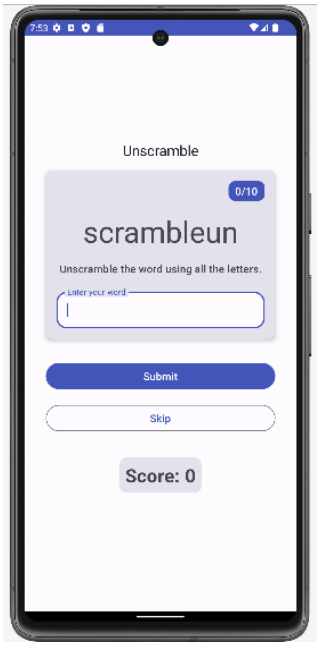


- **GameLayout** - mostra a funcionalidade principal do jogo, incluindo a palavra embaralhada, as instruções do jogo e um campo de texto que aceita os palpites do usuário.



Por fim, os elementos combináveis que usamos são:

- **GameScreen** - contém as funções combináveis *GameStatus* e *GameLayout*, o título do jogo, a contagem de palavras e os elementos combináveis dos botões *Submit* (Enviar) e *Skip* (Pular).



- **FinalScoreDialog** - mostra uma caixa de diálogo, ou seja, uma janela pequena que dá ao usuário opções para ***Play Again*** (Jogar novamente) ou ***Exit*** (Sair) do jogo.

5.2. Um pouco mais sobre arquitetura de apps

Baseando-se em princípios de arquitetura comum, cada app precisa ter pelo menos duas camadas:

- **Camada de interface:** uma camada que exibe os dados do app na tela, mas é independente dos dados, que é composta por:

  - **Elementos da interface:** componentes que renderizam os dados na tela. Crie esses elementos usando o Jetpack Compose.
  - **Detentores de estado:** componentes que armazenam os dados, os expõem à interface e processam a lógica do app. Um exemplo de detentor de estado é o **ViewModel**. Ao contrário da instância de atividade, os objetos **ViewModel** não são destruídos. O app retém automaticamente objetos **ViewModel** durante mudanças de configuração para que os dados retidos fiquem imediatamente disponíveis após a recomposição.
- **Camada de dados:** uma camada que armazena, recupera e expõe os dados do app.

5.3. Adicionando um ViewModel

Para resolver os problemas do código inicial que percebemos, vamos salvar os dados do jogo em um *ViewModel*.

- Abra **build.gradle.kts (Module :app)**, role até o bloco ***dependencies*** e adicione a seguinte dependência para *ViewModel*. Ela é usada para adicionar o modelo de visualização com reconhecimento de ciclo de vida ao app do *Compose*. Sincronize o projeto após esse passo.

```
implementation("androidx.lifecycle:lifecycle-viewmodel-
compose:2.6.1")
```

- Na pasta ui, crie uma classe/arquivo Kotlin com o nome **GameViewModel**. Estenda-a da classe ***ViewModel***.

```
import androidx.lifecycle.ViewModel

class GameViewModel : ViewModel() {
}
```

- Ainda na pasta ui, adicione uma classe de modelo para a interface do estado chamada **GameUiState**. Transforme-a em uma classe de dados e adicione uma variável para a palavra embaralhada atual.

```
data class GameUiState(
   val currentScrambledWord: String = ""
)
```

Um **StateFlow** (fluxo observável detentor de dados que emite as atualizações de estado novas e atuais) pode ser exposto no **GameUiState** para que os elementos combináveis possam detectar atualizações de estado da interface e fazer com que o estado da tela sobreviva às mudanças de configuração.

- Na classe **GameViewModel**, adicione a seguinte propriedade **_uiState**:

```
import kotlinx.coroutines.flow.MutableStateFlow

// Game UI state
private val _uiState = MutableStateFlow(GameUiState())
```

Uma propriedade de apoio permite que você retorne algo de um *getter* diferente do objeto exato. Para os métodos *getter* e *setter*, é possível substituir um deles, ou ambos, e fornecer um comportamento personalizado próprio, protegendo assim, o dado original. Isso é útil pois queremos evitar atualizações de estado vindas de outras classes.

- No arquivo **GameViewModel.kt**, adicione a **uiState** uma propriedade de apoio com o nome **_uiState**. Nomeie a propriedade como **uiState** e use o tipo **StateFlow<GameUiState>**.

```
import kotlinx.coroutines.flow.StateFlow

// Game UI state
private val _uiState = MutableStateFlow(GameUiState())
val uiState: StateFlow<GameUiState>
```

- Defina **uiState** como **_uiState.asStateFlow().**

```
import kotlinx.coroutines.flow.asStateFlow

// Game UI state
private val _uiState = MutableStateFlow(GameUiState())
val uiState: StateFlow<GameUiState> = _uiState.asStateFlow()
```

5.4. Exibindo palavra embaralhada aleatória

Agora vamos adicionar métodos auxiliares para escolher uma palavra aleatória do WordsData.kt e criar a palavra embaralhada.

- No **GameViewModel**, adicione uma propriedade com o nome **currentWord** do tipo String para salvar a palavra embaralhada atual.

```
private lateinit var currentWord: String
```

- Chame o arquivo de **pickRandomWordAndShuffle()** sem parâmetros de entrada e faça com que ele retorne uma String.

```
private lateinit var currentWord: String
```

- O AndroidStudio sinaliza alguns erros. Adicione a seguinte propriedade após a **currentWord** para servir como um conjunto mutável no armazenamento de palavras usadas no jogo:

```
private var usedWords: MutableSet<String> = mutableSetOf()
```

- Adicione outro método auxiliar para embaralhar a palavra atual com o nome **shuffleCurrentWord()**, que usa uma String e retorna a String embaralhada.

```
private fun shuffleCurrentWord(word: String): String {
    val tempWord = word.toCharArray()
    // Scramble the word
```

```
    tempWord.shuffle()
    while (String(tempWord).equals(word)) {
        tempWord.shuffle()
    }
    return String(tempWord)
}
```

- Adicione uma função auxiliar para inicializar o jogo com o nome **resetGame()**. Vamos usar essa função mais tarde para iniciar e reiniciar o jogo. Nessa função, limpe todas as palavras do conjunto **usedWords** e inicie o **_uiState**. Escolha uma nova palavra para **currentScrambledWord** usando **pickRandomWordAndShuffle**().

```
fun resetGame() {
    usedWords.clear()
    _uiState.value = GameUiState(currentScrambledWord =
pickRandomWordAndShuffle())
}
```

- Por fim, adicione um bloco init ao **GameViewModel** e chame o **resetGame()** dele.

```
init {
    resetGame(
)
}
```

Se executarmos nosso app agora, ainda não vão aparecer mudanças na interface. Ainda não estamos transmitindo os dados do *ViewModel* aos elementos combináveis na **GameScreen**.
Como as funções de composição aceitam estados e expõem eventos, um padrão de fluxo de dados unidirecional é adequado para o Jetpack Compose. Esse tipo de fluxo de dados é um padrão onde os estados fluem para baixo, e os eventos para cima. Ao seguir o fluxo de dados unidirecional, você pode dissociar os elementos que exibem o estado na interface das partes do app que armazenam e mudam o estado.

- Na função **GameScreen**, transmita um segundo argumento do tipo **GameViewModel** com um valor padrão de **viewModel().**

```
import
androidx.lifecycle.viewmodel.compose.viewModel

@Composable
fun GameScreen(
    gameViewModel: GameViewModel = viewModel()
) {
    // ...
}
```

- Na função **GameScreen()**, adicione uma nova variável com o nome **gameUiState**. Use o delegado **by** e chame **collectAsState()** no **uiState**. Essa abordagem garante que, sempre que haja uma mudança no valor do **uiState**, a recomposição ocorra para os elementos combináveis usando o valor **gameUiState**.

```
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue

@Composable
fun GameScreen(
    // ...
) {
    val gameUiState by
gameViewModel.uiState.collectAsState()
    // ...
}
```

- Transmita **gameUiState.currentScrambledWord** para o elemento combinável **GameLayout()**. Vamos adicionar o argumento em uma etapa posterior, então ignore o erro por enquanto.

```
GameLayout(
    currentScrambledWord =
gameUiState.currentScrambledWord,
    modifier = Modifier
        .fillMaxWidth()
        .wrapContentHeight()
        .padding(mediumPadding)
)
```

- Adicione **currentScrambledWord** como outro parâmetro à função combinável **GameLayout()**.

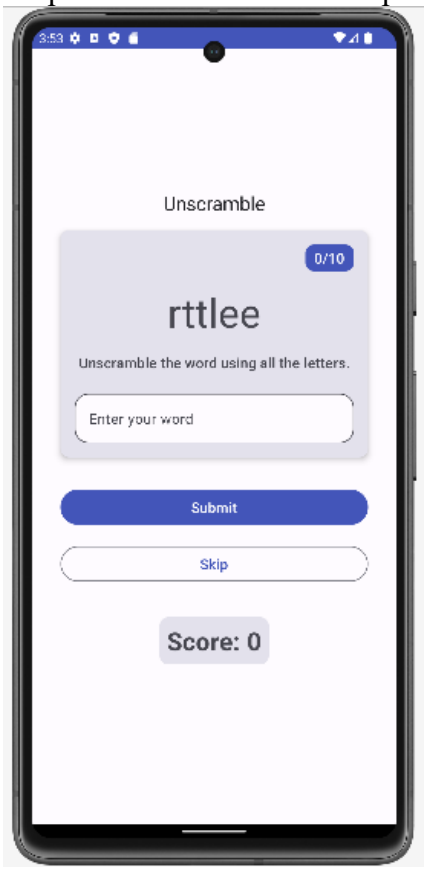
```
@Composable
fun GameLayout(currentScrambledWord: String, modifier: Modifier =
Modifier)
{
...
}
```

- Atualize a função de composição **GameLayout()** para mostrar **currentScrambledWord**. Defina o parâmetro **text** do segundo campo de texto na coluna como **currentScrambledWord**.

```
@Composable
fun GameLayout(
    // ...
) {
    Column(
        //...
    ) {
        Text(
```

```
            text = currentScrambledWord,
            style =
typography.displayMedium
        )

    //...
    }
}
```

- Execute e crie o app. A palavra embaralhada vai aparecer.



- Para mostrar a palavra do palpite, no arquivo **GameScreen.kt**, no elemento combinável **GameLayout()**, defina **onValueChange** como **onUserGuessChanged** e **onKeyboardDone()** como ação do teclado **onDone**. Não ligue para os erros por enquanto.

```
OutlinedTextField(
   value = "",
   singleLine = true,
   shape = shapes.large,
   modifier = Modifier.fillMaxWidth(),
   colors = TextFieldDefaults.colors(
       focusedContainerColor = colorScheme.surface,
       unfocusedContainerColor = colorScheme.surface,
```

```
        disabledContainerColor = colorScheme.surface,
    ),
    onValueChange = onUserGuessChanged,
    label = { Text(stringResource(R.string.enter_your_word)) }
},
    isError = false,
    keyboardOptions = KeyboardOptions.Default.copy(
        imeAction = ImeAction.Done
    ),
    keyboardActions = KeyboardActions(
        onDone = { onKeyboardDone() }
    )
)
```

- Na função combinável **GameLayout()**, adicione mais dois argumentos: o lambda **onUserGuessChanged** recebe um argumento **String** e não retorna nada, e **onKeyboardDone** não recebe nem retorna nada.

```
@Composable
fun GameLayout(
    onUserGuessChanged: (String) ->
Unit,
    onKeyboardDone: () -> Unit,
    currentScrambledWord: String,
    modifier: Modifier = Modifier
)
```

- Na chamada de função **GameLayout()**, adicione argumentos lambda para **onUserGuessChanged** e **onKeyboardDone**.

```
GameLayout(
    onUserGuessChanged = { gameViewModel.updateUserGuess(it)
},
    onKeyboardDone = { },
    currentScrambledWord = gameUiState.currentScrambledWord,
    modifier = Modifier
        .fillMaxWidth()
        .wrapContentHeight()
        .padding(mediumPadding)
)
```

Vamos definir o método **updateUserGuess** no **GameViewModel** em breve.

- No arquivo **GameViewModel.kt,** adicione um método com o nome **updateUserGuess()** que usa um argumento String, a palavra adivinhada pelo usuário. Dentro da função, atualize a **userGuess** usando a **guessedWord** transmitida.

```
fun updateUserGuess(guessedWord:
String){
    userGuess = guessedWord
}
```

- No arquivo **GameViewModel.kt**, adicione uma propriedade var chamada **userGuess**. Use **mutableStateOf()** para que o **Compose** observe esse valor e defina o valor inicial como " ".

```
import
androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue

var userGuess by mutableStateOf("")
   private set
```

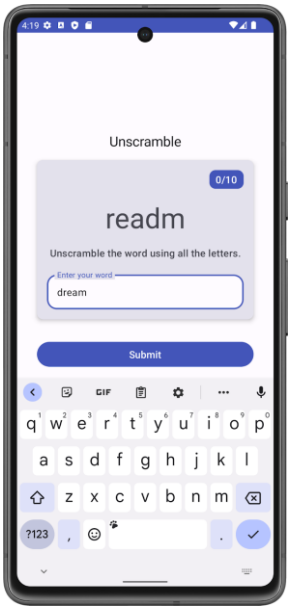
- No arquivo **GameScreen.kt**, em **GameLayout()**, adicione outro parâmetro String para **userGuess**. Defina o parâmetro value do **OutlinedTextField** como **userGuess**.

```
fun GameLayout(
   currentScrambledWord: String,
   userGuess: String,
   onUserGuessChanged: (String) -> Unit,
   onKeyboardDone: () -> Unit,
   modifier: Modifier = Modifier
) {
   Column(
       verticalArrangement =
Arrangement.spacedBy(24.dp)
   ) {
       //...
       OutlinedTextField(
           value = userGuess,
           //..
       )
   }
}
```

- Na função **GameScreen**, atualize a chamada de função **GameLayout()** para incluir o parâmetro **userGuess**.

```
GameLayout(
   currentScrambledWord = gameUiState.currentScrambledWord,
   userGuess = gameViewModel.userGuess,
   onUserGuessChanged = { gameViewModel.updateUserGuess(it)
},
   onKeyboardDone = { },
   //...
)
```

- Compile e execute o app. O campo de texto agora mostra o palpite do usuário.



5.5. Verificando a palavra adivinhada e atualizando a pontuação

No arquivo **GameViewModel**, adicione outro método com o nome **checkUserGuess()**. Na função **checkUserGuess()**, adicione um bloco if else para verificar se o palpite do usuário é igual à **currentWord**. Redefinir **userGuess** para uma string vazia.

```
fun checkUserGuess() {

    if (userGuess.equals(currentWord, ignoreCase = true))
{
    } else {
    }
    // Reset user guess
    updateUserGuess("")
}
```

Se o palpite do usuário estiver errado, defina **isGuessedWordWrong** como true. **MutableStateFlow<T>**. update() atualiza o **MutableStateFlow.value** usando o valor especificado (links em inglês).

```
import kotlinx.coroutines.flow.update

if (userGuess.equals(currentWord, ignoreCase = true))
{
} else {
    // User's guess is wrong, show an error
    _uiState.update { currentState ->
        currentState.copy(isGuessedWordWrong = true)
    }
}
```

Na classe **GameUiState**, adicione um **Boolean** chamado **isGuessedWordWrong** e inicialize-o como **false**.

```
data class GameUiState(
    val currentScrambledWord: String = "",
    val isGuessedWordWrong: Boolean =
false,
)
```

No arquivo **GameScreen.kt**, no final da função combinável **GameScreen()**, chame **gameViewModel.checkUserGuess()** dentro da expressão lambda **onClick** do botão Submit. Esse passo serve para definir o erro no campo de texto.

```
Button(
    modifier = Modifier.fillMaxWidth(),
    onClick = { gameViewModel.checkUserGuess()
}
) {
    Text(
        text = stringResource(R.string.submit),
        fontSize = 16.sp
    )
}
```

Na função combinável **GameScreen()**, atualize a chamada de função **GameLayout()** para transmitir **gameViewModel.checkUserGuess()** na expressão lambda **onKeyboardDone**.

```
GameLayout(
    onUserGuessChanged = { gameViewModel.updateUserGuess(it) },
    userGuess = gameViewModel.userGuess,
    onKeyboardDone = { gameViewModel.checkUserGuess() },
    currentScrambledWord = gameUiState.currentScrambledWord,
    modifier = Modifier
        .fillMaxWidth()
        .wrapContentHeight()
        .padding(mediumPadding)
)
```

Na função de composição **GameLayout()**, adicione um parâmetro de função para Boolean, **isGuessWrong**. Defina o parâmetro **isError** do **OutlinedTextField** como **isGuessWrong** para mostrar o erro no campo de texto quando o palpite do usuário estiver errado.

```
fun GameLayout(
    currentScrambledWord: String,
    isGuessWrong: Boolean,
    userGuess: String,
    onUserGuessChanged: (String) -> Unit,
    onKeyboardDone: () -> Unit,
    modifier: Modifier = Modifier
) {
    Column(
        // ,...
        OutlinedTextField(
            // ...
            isError = isGuessWrong,
            keyboardOptions =
KeyboardOptions.Default.copy(
                imeAction = ImeAction.Done
```

```
            ),
            keyboardActions = KeyboardActions(
                onDone = { onKeyboardDone() }
            ),
        )
}
}
```

Na função combinável **GameScreen()**, atualize a chamada de função **GameLayout()** para transmitir **isGuessWrong**.

```
GameLayout(
    onUserGuessChanged = { gameViewModel.updateUserGuess(it) },
    userGuess = gameViewModel.userGuess,
    onKeyboardDone = { gameViewModel.checkUserGuess() },
    currentScrambledWord = gameUiState.currentScrambledWord,
    isGuessWrong = gameUiState.isGuessedWordWrong,
    modifier = Modifier
        .fillMaxWidth()
        .wrapContentHeight()
        .padding(mediumPadding)
)
```

Compile e execute o app. Insira um palpite errado e clique em Submit. Observe que o campo de texto fica vermelho, indicando o erro.

Agora vamos atualizar a pontuação e a contagem de palavras enquanto o usuário joga.

No **GameUiState**, adicione uma variável score e a inicialize em zero.

```
data class GameUiState(
    val currentScrambledWord: String = "",
    val isGuessedWordWrong: Boolean = false,
    val score: Int = 0
)
```

Para atualizar o valor da pontuação, aumente o valor de score na função **checkUserGuess()**, em **GameViewModel**, dentro da condição if para quando o palpite do usuário estiver correto.

```
fun checkUserGuess() {

    if (userGuess.equals(currentWord, ignoreCase = true)) {
        val updatedScore =
_uiState.value.score.plus(SCORE_INCREASE)
    } else {
        ...
        }
    }
 }
```

No arquivo **GameViewModel**, adicione outro método com o nome **updateGameState** para atualizar a pontuação, incrementar a contagem de palavras atual e escolher uma nova palavra no arquivo **WordsData.kt**. Adicione uma Int chamada **updatedScore** como parâmetro. Atualize as variáveis da interface do estado do jogo desta forma:

```
private fun updateGameState(updatedScore: Int) {
    _uiState.update { currentState ->
        currentState.copy(
            isGuessedWordWrong = false,
            currentScrambledWord = pickRandomWordAndShuffle(),
            score = updatedScore
        )
    }
}
```

Na função **checkUserGuess()**, se o palpite do usuário estiver correto, faça uma chamada para **updateGameState** com a pontuação atualizada para preparar o jogo para a próxima rodada.

```
fun checkUserGuess() {

    if (userGuess.equals(currentWord, ignoreCase = true)) {
        val updatedScore =
_uiState.value.score.plus(SCORE_INCREASE)
        updateGameState(updatedScore)
    }else{
      //. . .
    }
}
```

A **checkUserGuess()** concluída vai ficar assim:

```
fun checkUserGuess() {

    if (userGuess.equals(currentWord, ignoreCase = true)) {
        val updatedScore =
_uiState.value.score.plus(SCORE_INCREASE)
        updateGameState(updatedScore)
    } else {
        _uiState.update { currentState ->
            currentState.copy(isGuessedWordWrong = true)
        }
    }
    updateUserGuess("")
}
```

Em seguida, assim como as atualizações da pontuação, vai ser necessário atualizar a contagem de palavras. Adicione outra variável para a contagem em **GameUiState**. Chame-a de **currentWordCount** e inicialize em 1.

```
data class GameUiState(
    val currentScrambledWord: String = "",
    val currentWordCount: Int = 1,
    val isGuessedWordWrong: Boolean = false,
    val score: Int = 0
)
```

No arquivo **GameViewModel.kt**, na função **updateGameState()**, aumente a contagem de palavras, como mostrado abaixo. A função **updateGameState()** é chamada para preparar o jogo para a próxima rodada.

```
private fun updateGameState(updatedScore: Int) {
    _uiState.update { currentState ->
        currentState.copy(
            isGuessedWordWrong = false,
            currentScrambledWord = pickRandomWordAndShuffle(),
            currentWordCount =
currentState.currentWordCount.inc(),
            score = updatedScore
        )
    }
}
```

Agora precisamos transmitir dados de pontuação e contagem de palavras de **ViewModel** para **GameScreen**. No arquivo **GameScreen.kt** da função combinável **GameLayout()**, adicione a contagem de palavras como argumento e transmita os argumentos do formato **wordCount** para o elemento de texto.

```
fun GameLayout(
    onUserGuessChanged: (String) -> Unit,
    onKeyboardDone: () -> Unit,
    wordCount: Int,
    //...
) {
    //...
```

```
    Card(
        //...
    ) {
        Column(
            // ...
        ) {
            Text(
                //..
                text = stringResource(R.string.word_count,
wordCount),
                style = typography.titleMedium,
                color = colorScheme.onPrimary
            )
     // ...
}
```

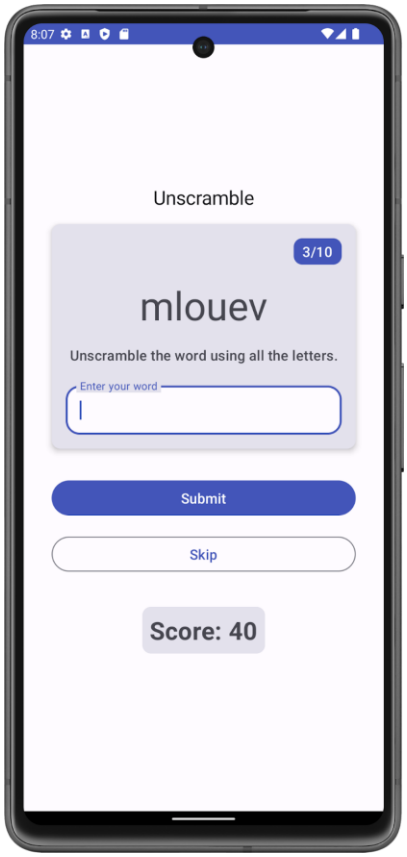
Atualize a chamada de função **GameLayout()** para incluir a contagem de palavras.

```
GameLayout(
    onUserGuessChanged = { gameViewModel.updateUserGuess(it) },
    userGuess = gameViewModel.userGuess,
    //...
    )
```

Na função combinável **GameScreen()**, atualize a chamada de função **GameStatus()** para incluir os parâmetros score. Transmita a pontuação do **gameUiState**.

```
GameStatus(score=gameUiState.score,
modifier=Modifier.padding(20.dp))
```

Crie e execute o app. Insira um palpite para a palavra e clique em Submit. A pontuação e a contagem de palavras são atualizadas. Clique em Skip e observe que nada acontece.



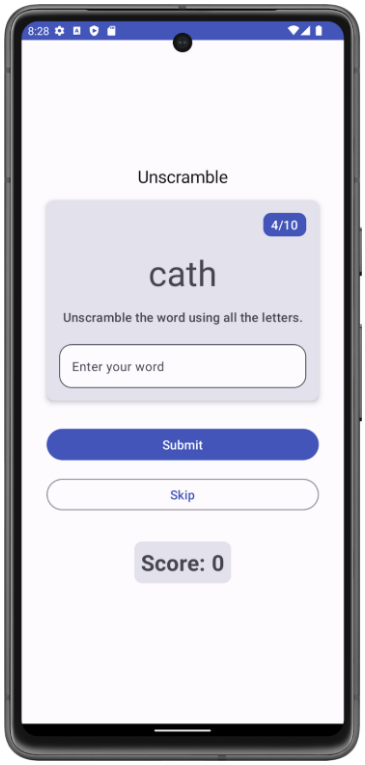
Para implementar o **Skip**, precisamos transmitir o callback do evento de pulo para o **GameViewModel**. Para isso, no arquivo **GameScreen.kt**, na função de composição **GameScreen()**, faça uma chamada para **gameViewModel.skipWord()** na expressão lambda onClick.

```
OutlinedButton(
    onClick = { gameViewModel.skipWord() },
    modifier = Modifier.fillMaxWidth()
)
```

Em seguida, em **GameViewModel.kt**, adicione o método **skipWord()**. Na função **skipWord()**, faça uma chamada para **updateGameState()**, transmitindo a pontuação e redefina o palpite do usuário.

```
OutlinedButton(
    onClick = { gameViewModel.skipWord() },
    modifier = Modifier.fillMaxWidth()
)
```

Agora execute o app e jogue uma partida, você conseguirá pular palavras! Ainda não limitamos o número de partidas, então você conseguirá jogar mais de 10.



5.6. Processando a última rodada do jogo

Para implementar a lógica de fim de jogo, primeiro é preciso verificar se o usuário atingiu o número máximo de palavras. No **GameViewModel**, adicione um bloco **if-else** e mova o corpo da função existente para o bloco **else**. Adicione uma condição **if** para verificar se o tamanho das **usedWords** é igual a **MAX_NO_OF_WORDS.**

No bloco if, adicione a sinalização Boolean **isGameOver** e a defina como true para indicar o fim do jogo. Atualize a score e redefina **isGuessedWordWrong** dentro do bloco if. O código abaixo mostra como a função vai ficar.

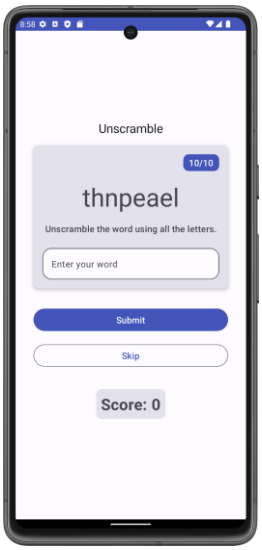
```
private fun updateGameState(updatedScore: Int) {
    if (usedWords.size == MAX_NO_OF_WORDS){
        //Last round in the game, update isGameOver to true,
don't pick a new word
        _uiState.update { currentState ->
            currentState.copy(
                isGuessedWordWrong = false,
                score = updatedScore,
                isGameOver = true
            )
        }
    } else{
        // Normal round in the game
        _uiState.update { currentState ->
            currentState.copy(
                isGuessedWordWrong = false,
                currentScrambledWord =
```

```
pickRandomWordAndShuffle(),
                currentWordCount =
currentState.currentWordCount.inc(),
                score = updatedScore
            )
        }
    }
}
```

No **GameUiState**, adicione a variável Boolean **isGameOver** e a defina como **false**.

```
data class GameUiState(
    val currentScrambledWord: String = "",
    val currentWordCount: Int = 1,
    val isGuessedWordWrong: Boolean = false,
    val score: Int = 0,
    val isGameOver: Boolean = false
)
```

Execute o app e jogue uma partida. Não será possível jogar mais de 10 palavras.



Agora precisamos informar ao usuário que o jogo terminou e perguntar se ele quer jogar de novo. Para isso criaremos uma caixa de diálogo, que são pequenas janelas que levam o usuário a tomar uma decisão ou inserir mais informações.
O arquivo **GameScreen.kt** no código inicial já fornece uma função que mostra uma caixa de diálogo de alerta com opções para sair ou reiniciar o jogo, chamada **FinalScoreDialog**. Precisamos fazer ela ser chamada quando o jogo realmente é finalizado.

No arquivo **GameScreen.kt**, no final da função combinável **GameScreen()**, depois do bloco Column, adicione uma condição if para conferir **gameUiState.isGameOver**.

No bloco if, exiba a caixa de diálogo de alerta. Faça uma chamada para **FinalScoreDialog()** transmitindo score e **gameViewModel.resetGame()** para o callback de evento **onPlayAgain**.

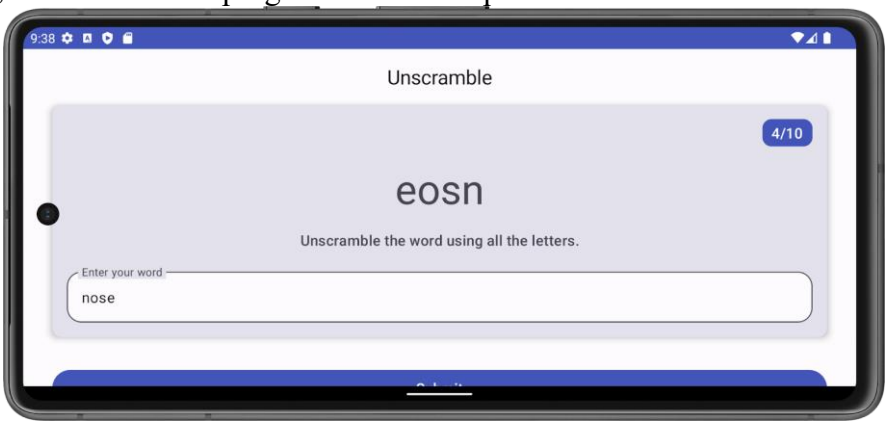
```
Column(
     //...
    )

{
    if (gameUiState.isGameOver) {
    FinalScoreDialog(
        score = gameUiState.score,
        onPlayAgain = { gameViewModel.resetGame() }
    )
    }
```

No arquivo **GameViewModel.kt**, chame a função **resetGame()**, inicialize **_uiState** e escolha uma nova palavra.

```
fun resetGame() {
    usedWords.clear()
    _uiState.value = GameUiState(currentScrambledWord =
pickRandomWordAndShuffle())
}
```

Compile e execute o app. Jogue até o final e observe a caixa de diálogo de alerta com as opções Sair ou Jogar novamente. Teste as opções mostradas na caixa de diálogo de alerta.

O *ViewModel* armazena os dados relacionados ao app que não são destruídos quando o framework do Android destrói e recria a atividade. Os objetos *ViewModel* são retidos automaticamente e não são destruídos, como a instância de atividade, durante a mudança da configuração. Os dados retidos ficam imediatamente disponíveis após a recomposição. Por isso, se você mudar a orientação do dispositivo no meio de uma partida, os dados do seu progresso não serão perdidos.



Referência

Documentação oficial de Kotlin para Android. Noções básicas do Android com o Compose. Google. Acesso em 29 de agosto de 2023. URL: https://developer.android.com/courses/android-basics-compose/course?gclid=CjwKCAjwivemBhBhEiwAJxNWN25rWjfhk8Hx_7zs43wqKEgN1FoY9kkzSd4u6rlzSkx52TmZ9iuTxxoCcc0QAvD_BwE&gclsrc=aw.ds&hl=pt-br

Documentação oficial de Kotlin para Android. Conceitos básicos do Kotlin para Android. Google. Acesso em 29 de agosto de 2023. URL: https://developer.android.com/courses/kotlin-android-fundamentals/overview?hl=pt-br

Documentação oficial de Kotlin para Android. Android para desenvolvedores. Google. Acesso em 29 de agosto de 2023. URL: https://developer.android.com/?hl=pt-br

Curso de formação em desenvolvimento Android. Desenvolvido por Paulo Salvatore, Globalcode e Movile. Julho de 2022. URL: https://developer.android.com/?hl=pt-br