

Unidade II

Nesta unidade, exploraremos detalhadamente o ambiente de desenvolvimento, o Android Studio. Conheceremos as ferramentas e as técnicas essenciais para criar aplicativos, utilizando as tradicionais views e o moderno Jetpack Compose.

Primeiramente, abordaremos a configuração do Android Studio, incluindo a instalação e a configuração inicial do ambiente. Em seguida, aprenderemos a navegar por sua interface, explorando suas principais funcionalidades, como o editor de código, o emulador de dispositivos e as ferramentas de depuração.

Depois de estabelecermos uma base sólida no uso do Android Studio, estudaremos as técnicas de desenvolvimento de interfaces de usuário. Veremos como criar layouts utilizando views, compreendendo a hierarquia de visualização e os diferentes tipos de layouts disponíveis. Posteriormente, introduziremos o Jetpack Compose, uma ferramenta declarativa para a construção de interfaces de usuário no Android. Compararemos as abordagens baseadas em views e em Jetpack Compose, destacando as vantagens e as melhores práticas de cada uma.

5 ANDROID STUDIO

O Android Studio é o IDE oficial para a criação de aplicativos Android, desenvolvido e distribuído pelo Google. Um IDE oferece um conjunto de ferramentas que permitem aos desenvolvedores projetar, criar, executar e testar software, especificamente aplicativos para a plataforma Android. Baseado no IntelliJ IDEA, o Android Studio vem com o plug-in do Android pré-instalado, além de várias modificações específicas para a plataforma Android.

Para utilizar o Android Studio, é necessário um computador com Linux, macOS (10.14 Mojave ou mais recente), ChromeOS ou uma versão de 64 bits do Windows (8, 10 ou 11), além de acesso à internet. Os requisitos do sistema para o Android Studio no Windows incluem Microsoft Windows 8/10/11 de 64 bits, arquitetura de CPU x86_64, Intel Core de segunda geração ou mais recente, ou CPU AMD com suporte a hipervisor do Windows. É necessário ter pelo menos 8 GB de RAM livre durante a execução do sistema e um mínimo de 8 GB de espaço em disco disponível (para o ambiente de desenvolvimento integrado, SDK do Android e Android Emulator). A resolução de tela mínima recomendada é de 1.280 x 800.

5.1 Instalação



Saiba mais

Para baixar o Android Studio, basta acessar o site oficial do desenvolvedor Android:

DEVELOPERS. *Android Studio*. [s.d.]a. Disponível em: <https://shre.ink/M6h0>. Acesso em: 6 mar. 2025.

O sistema operacional será detectado automaticamente e poderá iniciar o download. Caso o link não funcione, pesquise Android Studio download em qualquer mecanismo de busca.

Ao encontrar a página, siga os seguintes passos:

- Clique em fazer o download do Android Studio Ladybug.

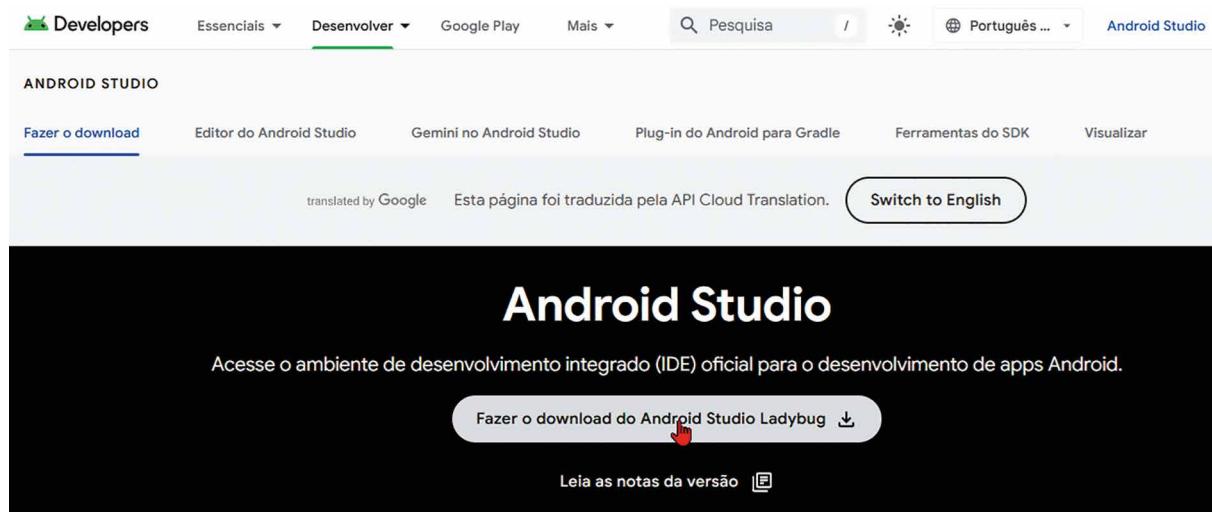


Figura 51 – Site do developers para baixar o Android Studio

Disponível em: <https://shre.ink/M6h0>. Acesso em: 6 mar. 2025.

- A página de termos e condições com o contrato de licença do Android Studio será aberta (figura 52). Leia o contrato de licença. Caso concorde com termos e condições, marque a caixa de seleção "Li e concordo com os Termos e Condições acima" na parte inferior da página. Clique em "Fazer o download do Android Studio" para iniciar o download. Quando solicitado, salve o arquivo em um local de fácil acesso, como a pasta Downloads.

Fazer o download Android Studio Meerkat | 2024.3.1

Antes de fazer o download, é necessário concordar com os Termos e Condições a seguir.

Termos e Condições

Este é o Contrato de Licença do kit de desenvolvimento de software do Android.

1. Introdução

1.1 O kit de desenvolvimento de software do Android (referido no Contrato de Licença como "SDK" e incluindo especificamente os arquivos do sistema Android, pacotes de APIs e complementos de APIs do Google) é licenciado para você de acordo com os termos do Contrato de Licença. O Contrato de Licença constitui um acordo legal entre você e o Google em relação ao uso do SDK. 1.2 "Android" significa a pilha de software Android para dispositivos, conforme disponibilizado no Android Open Source Project, localizado no seguinte URL: <https://source.android.com/>, atualizado periodicamente. 1.3 Uma "implementação compatível" significa qualquer dispositivo Android que (i) esteja em conformidade com o documento de definição de compatibilidade do Android, que pode ser encontrado no site de compatibilidade do Android (<https://source.android.com/compatibility>) e atualizado periodicamente; e (ii) seja aprovado no conjunto de teste de compatibilidade do Android (CTS). 1.4 "Google" se refere à Google LLC, empresa estabelecida de acordo com as leis do estado de Delaware, EUA, e que opera de acordo com as leis dos EUA, com sede principal em 1600 Amphitheatre Parkway, Mountain View, CA 94043, EUA.

2. Aceitação deste Contrato de Licença

Contrato de Licença não podem ser atribuídos ou transferidos para terceiros. Se você concorda com os termos e condições do Contrato de Licença, clique em "Aceito". Caso contrário, não pode usar o software. Clique em "Cancelar" para sair da página. Ao clicar em "Aceito", você concorda com os termos e condições do Contrato de Licença, que são regidos pelas leis do estado da Califórnia sem considerar conflitos de provisões legais. Você e o Google concordam em se submeterem à competência exclusiva dos tribunais localizados no condado de Santa Clara, Califórnia, para resolver qualquer questão legal decorrente do Contrato de Licença. Não obstante, você aceita que o Google continuará a ter o direito de solicitar as medidas judiciais de reparação cabíveis (ou tipo equivalente de medida jurídica urgente) em qualquer jurisdição. 27 de julho de 2021

Li e aceito os Termos e Condições acima

[Fazer o download de Android Studio Meerkat | 2024.3.1 para Windows](#)

android-studio-2024.3.1.13-windows.exe

Figura 52 – Termos e condições

Disponível em: <https://shre.ink/M6uK>. Acesso em: 6 mar. 2025.

Após o download, vá até o local onde o arquivo foi salvo e inicie o processo de instalação.



Figura 53 – Início do processo de instalação

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

À medida que as telas aparecerem, clique em Next ou Confirm.

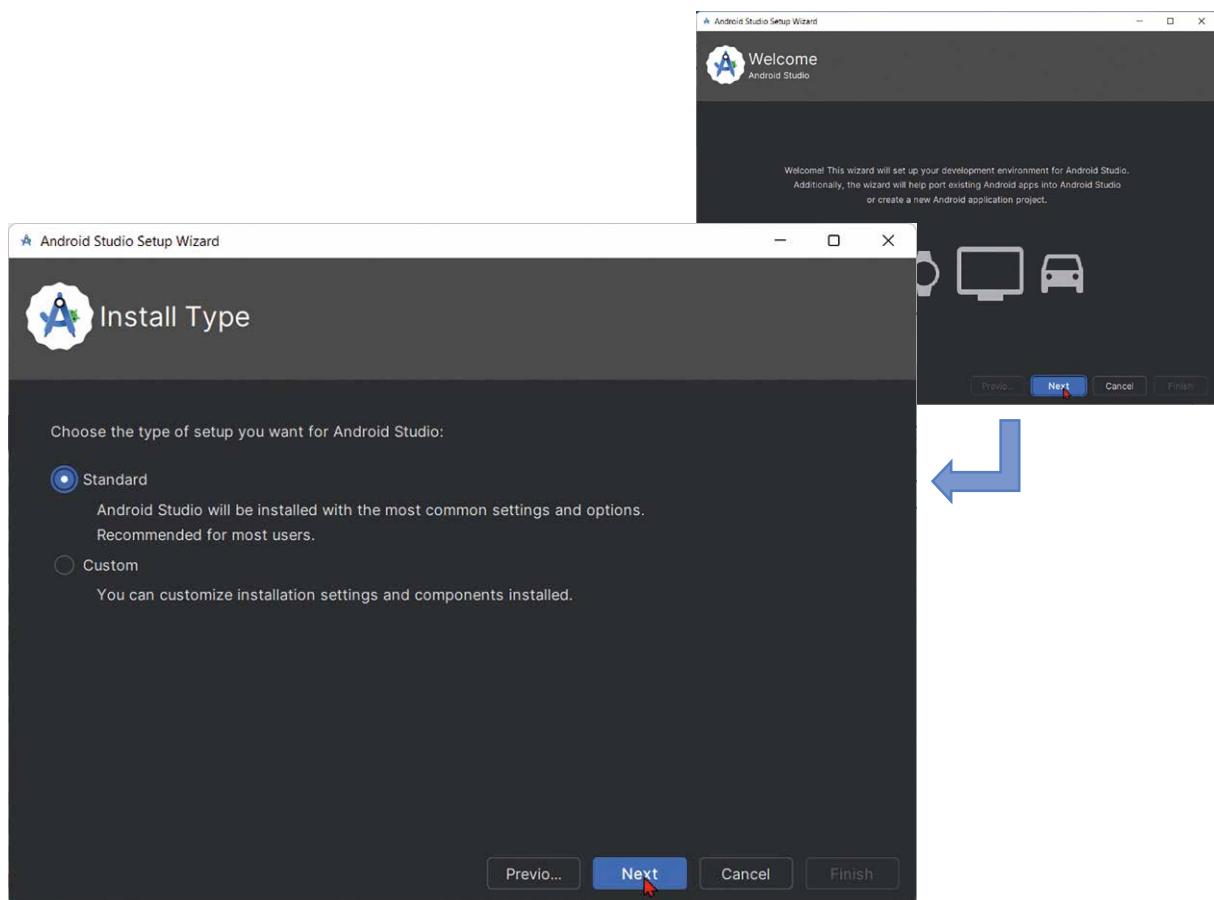


Figura 54 – Escolher o tipo de instalação

A menos que deseje realizar uma instalação personalizada removendo bibliotecas e funcionalidades, escolha a opção Standard (figura 54).

Uma vez escolhido o tipo de instalação, muitos componentes Software Development Kit (Kit de Desenvolvimento de Software) (SDK) necessitam aceitar a licença de uso. Uma vez aceita, o sistema de instalação baixará os componentes atualizados (figura 55).

Unidade II

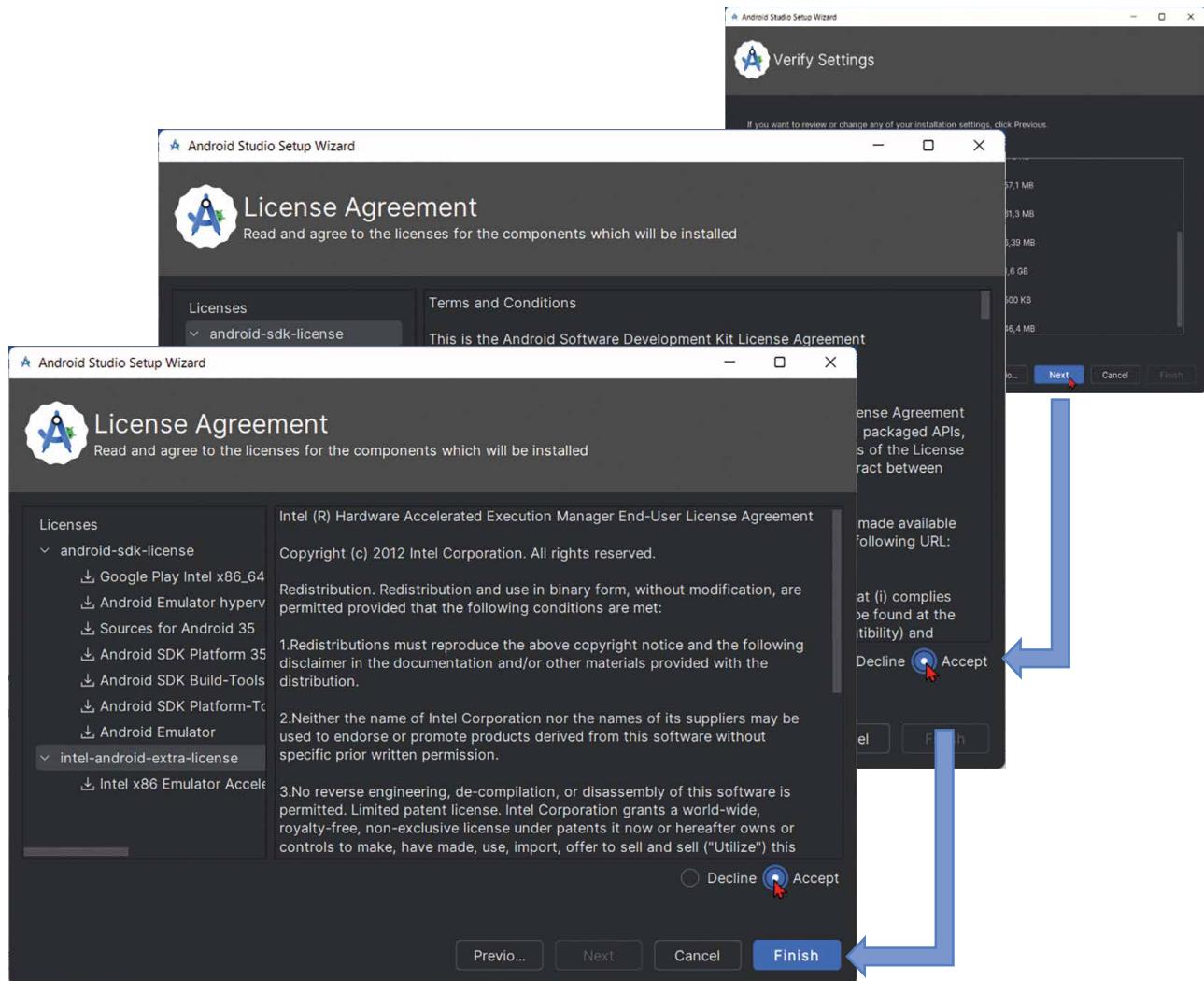


Figura 55 – Instalações das SDK

O processo demorará um pouco, pois serão baixados muitos arquivos. Ao terminar, o botão Finish será ativado.

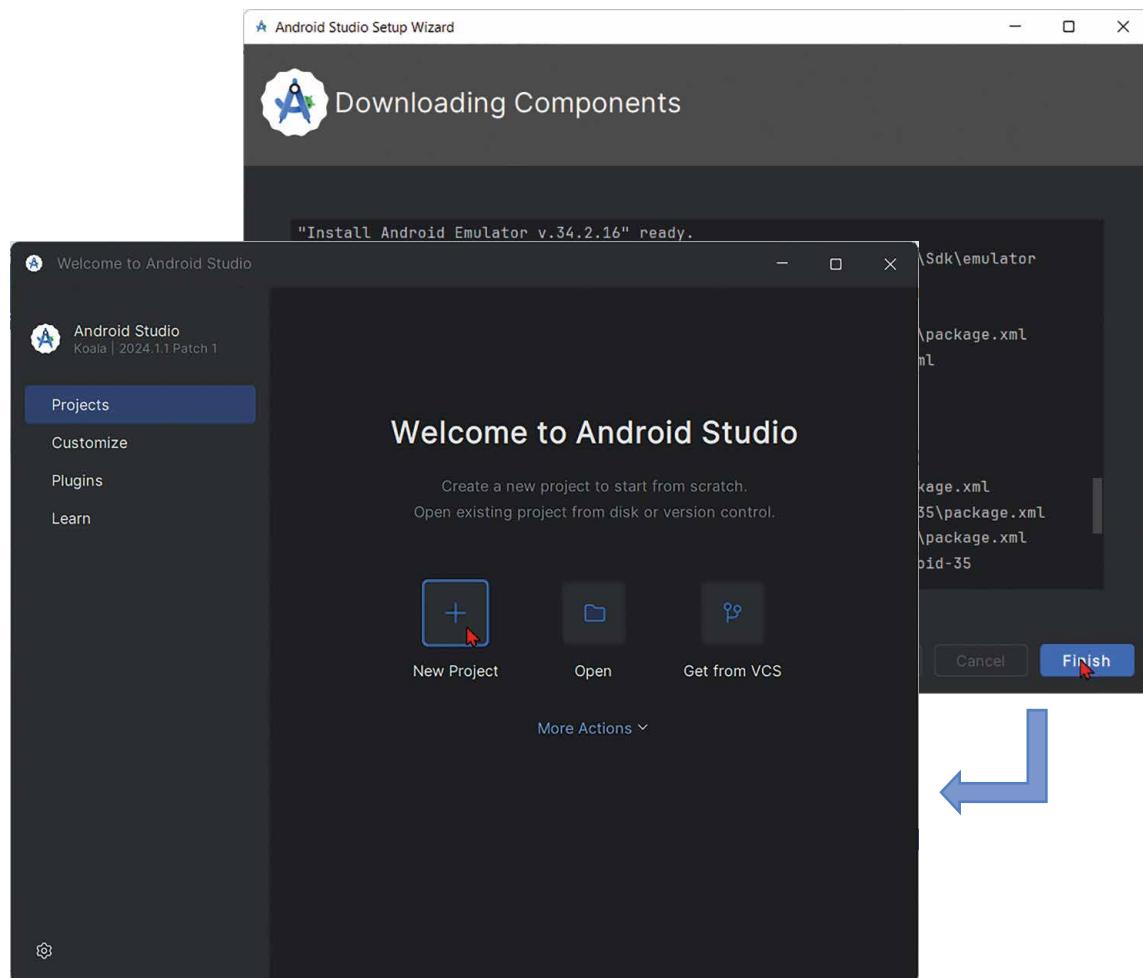


Figura 56 – Encerramento da instalação e execução do Android Studio

5.2 Iniciando um projeto

A primeira tela que o ambiente do Android Studio mostra é "Welcome to Android Studio" (figura 57). A tela agora está com o tema claro. Ela será adotada, pois permite melhor visibilidade do texto impresso. No Android Studio, pode continuar com o tema escuro.



Esta alteração foi feita entrando no item *Customize* e alterando o item *Appearance* para *theme Light*.

Unidade II

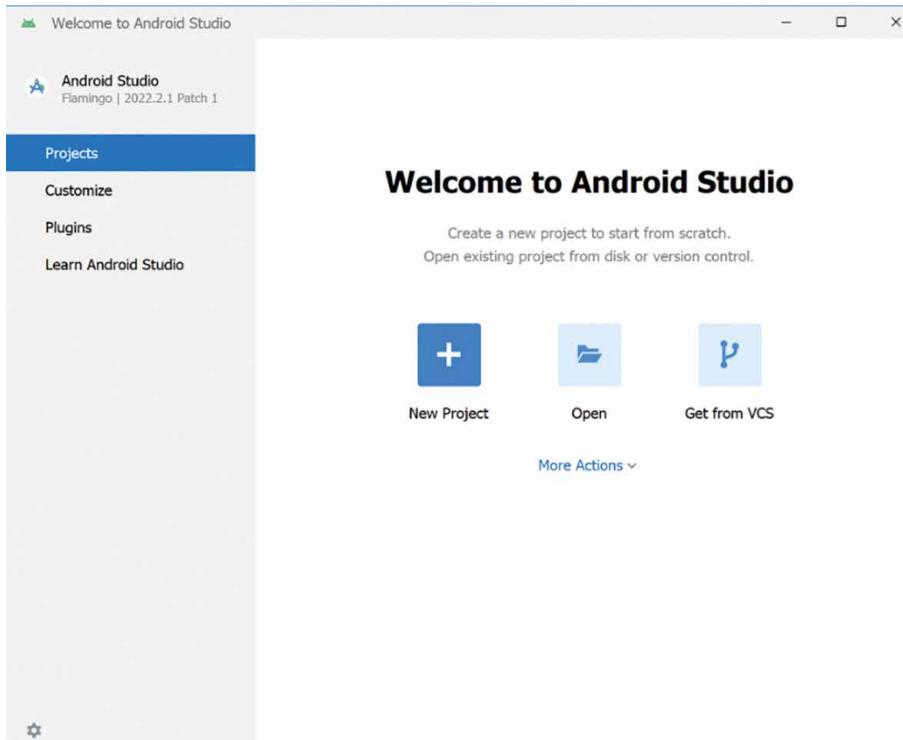


Figura 57 – Tela inicial do Welcome to Android Studio

Clique em New Project. A tela com os modelos (template) iniciais é carregada.

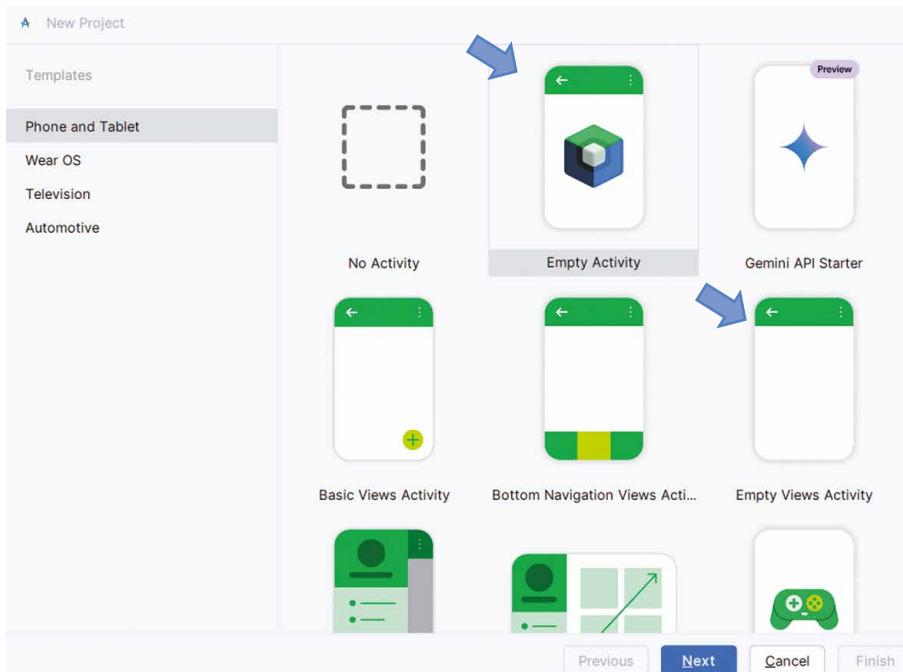


Figura 58 – Tela de definição do tipo de template do projeto que será desenvolvido

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Na figura 58, observamos dois templates indicados. O primeiro é o Empty Activity, que carregará um projeto a ser desenvolvido utilizando o Jetpack Compose como base. O segundo é o Empty Views Activity, que carregará um projeto a ser desenvolvido utilizando a técnica das views. São esses templates que utilizaremos neste livro-texto.

Escolha o Empty Activity e clique em Next. Ele abrirá a tela de configuração inicial do seu projeto.

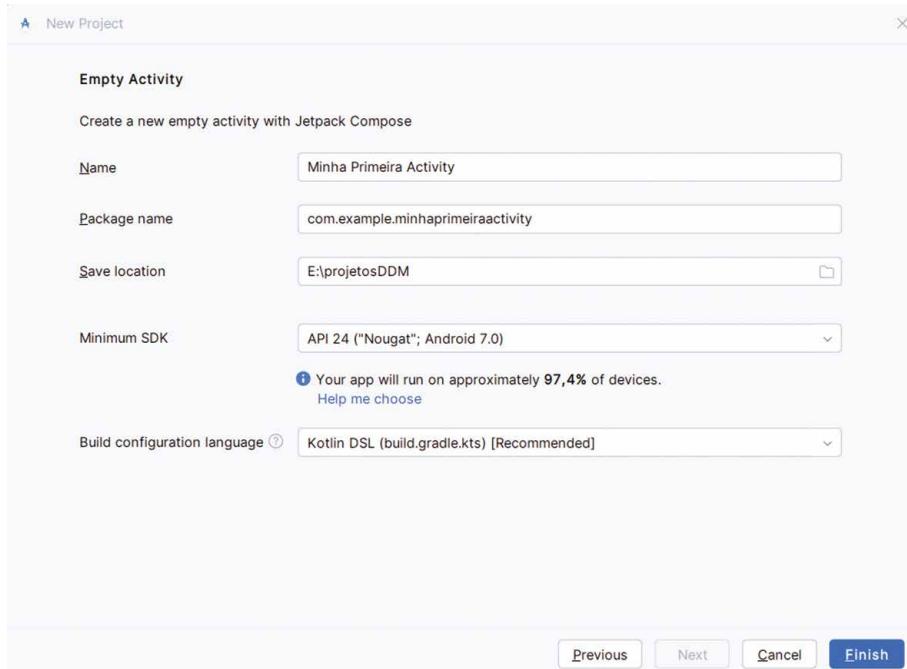


Figura 59

O campo Nome é utilizado para inserir o nome do seu projeto. Para esse codelab, digite Minha Primeira Activity.

Deixe o campo Package name como está. Ele define como seus arquivos serão organizados na estrutura de diretórios. Nesse caso, o nome do pacote será com.example.minhaprimeiraactivity.

Mantenha o campo Save location como está. Ele especifica o local onde todos os arquivos relacionados ao seu projeto serão salvos. Anote esse local no computador para encontrar seus arquivos facilmente.

Mantenha o campo MinimumSDK como está, pois indica a versão mínima do Android na qual o aplicativo pode ser executado. No caso, API 24: Android 7.0 (Nougat).

Finalmente, no campo Build configuration language também mantenha o campo já preenchido Kotlin DSL (build.gradle.kts) [Recommended]. Ele indica a linguagem para o desenvolvimento do seu projeto. Essa tela é comum a ambos os templates que iremos utilizar. Clique em Finish. Agora é preciso esperar um bom tempo até a montagem do projeto terminar.

Unidade II

Importing 'projetosDDM' Gradle Project

Figura 60 – Término da inicialização

Conforme o computador e a instalação, o Android Studio estará pronto para ser utilizado (figura 61). Verifique em toda a moldura se não há nenhuma barra de carregamento ou algo rodando.

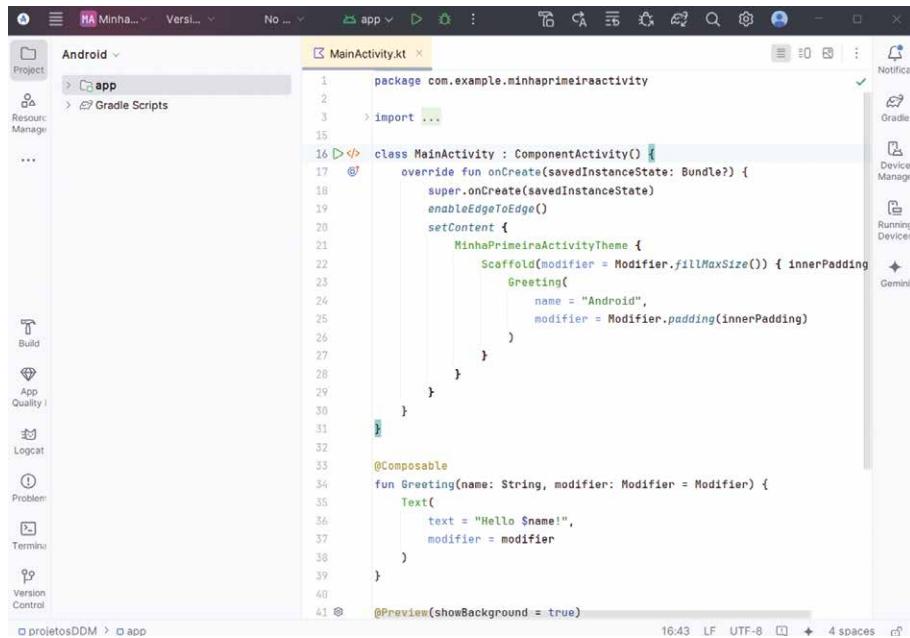


Figura 61 – Tela inicial após o carregamento total

Se a tela de design não aparecer à direita, clique no botão de layout do editor e selecione Design (figura 62). Se já estiver visível, não é necessário fazer nenhuma alteração.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

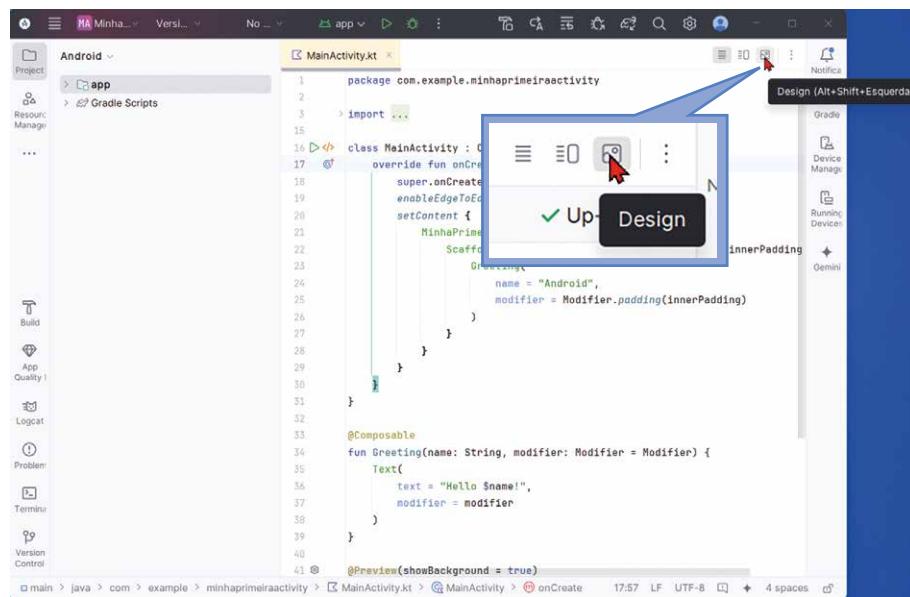


Figura 62 – Controle do layout do editor

A pré-visualização (preview) estará inicialmente desativada. Para ativá-la, clique em Build & Refresh (figura 63) para realizar uma compilação completa do projeto, transformando o código-fonte em um aplicativo executável. Após a construção bem-sucedida, a pré-visualização será atualizada para refletir as mudanças feitas no código, permitindo ver instantaneamente como as alterações visuais afetam a interface do usuário do seu aplicativo.

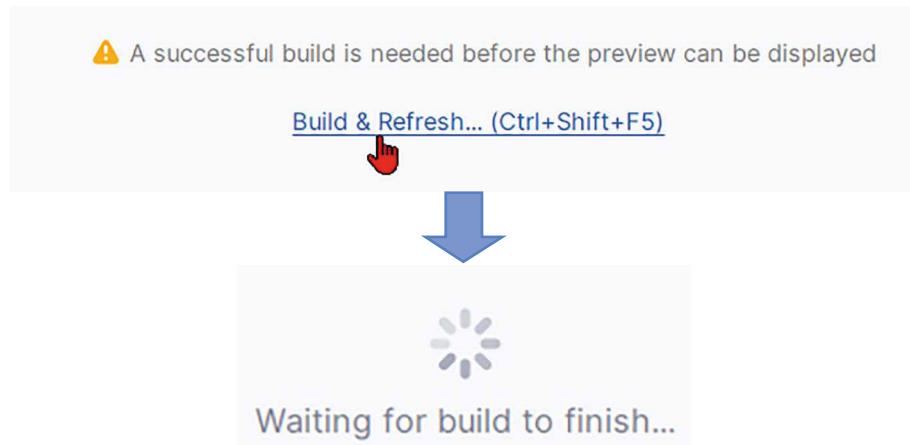


Figura 63 – Iniciando a construção do aplicativo para realizar o preview



Ocasionalmente, pode ocorrer uma falha na construção devido a uma biblioteca desatualizada. Nesse caso, siga as instruções fornecidas pelo próprio Android Studio.

Unidade II

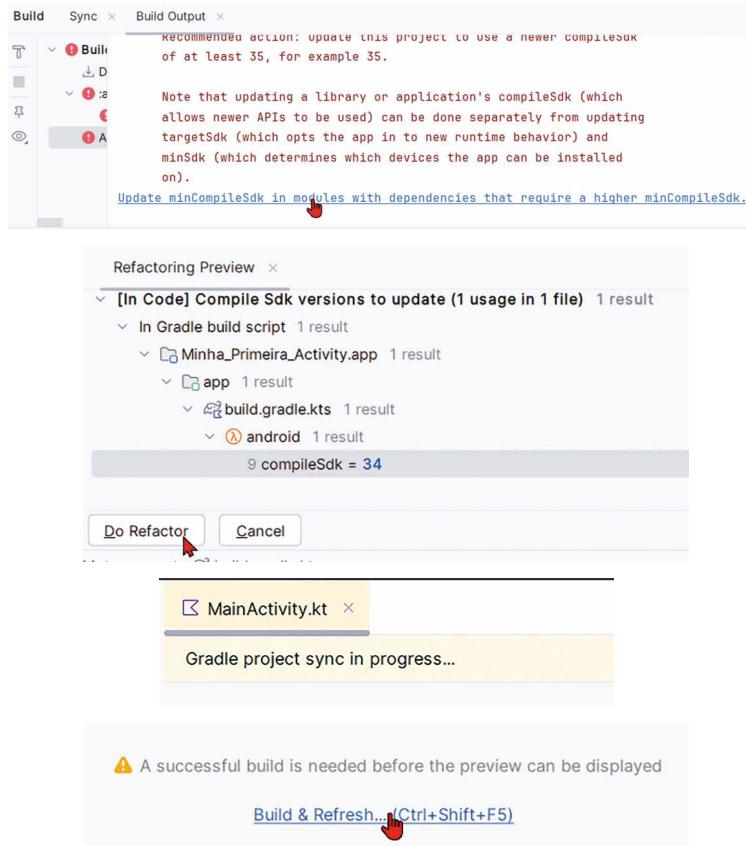


Figura 64

Espere a construção terminar e o estado mudar para Up-to-date.

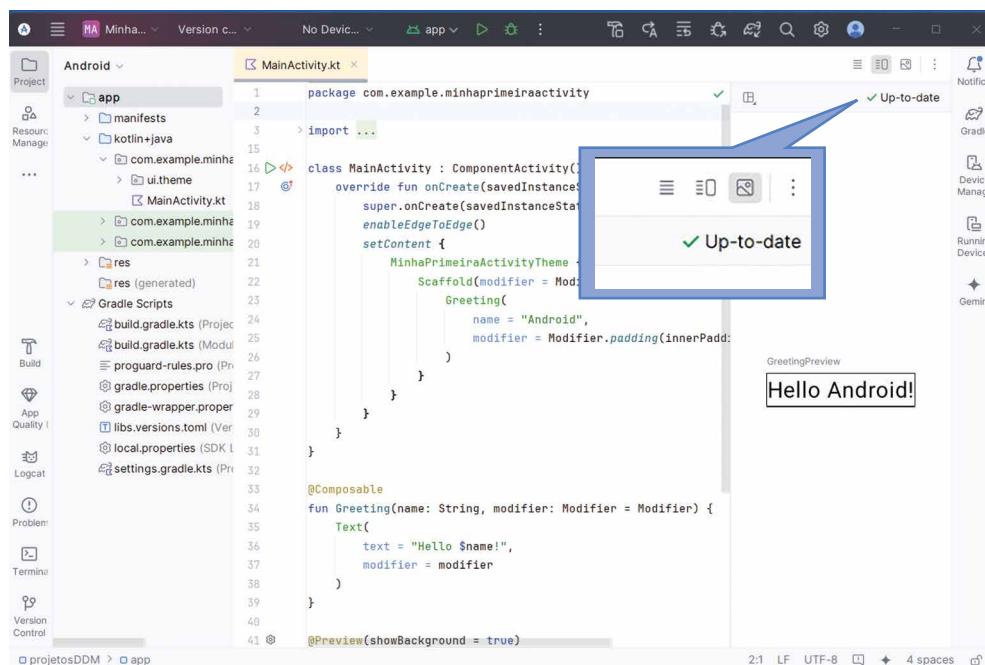


Figura 65 – Término da construção

5.3 Interface

Com tudo carregado e construído entenderemos a interface do editor. Para deixar o editor na visualização padrão, clique em Split no botão de layout do editor.

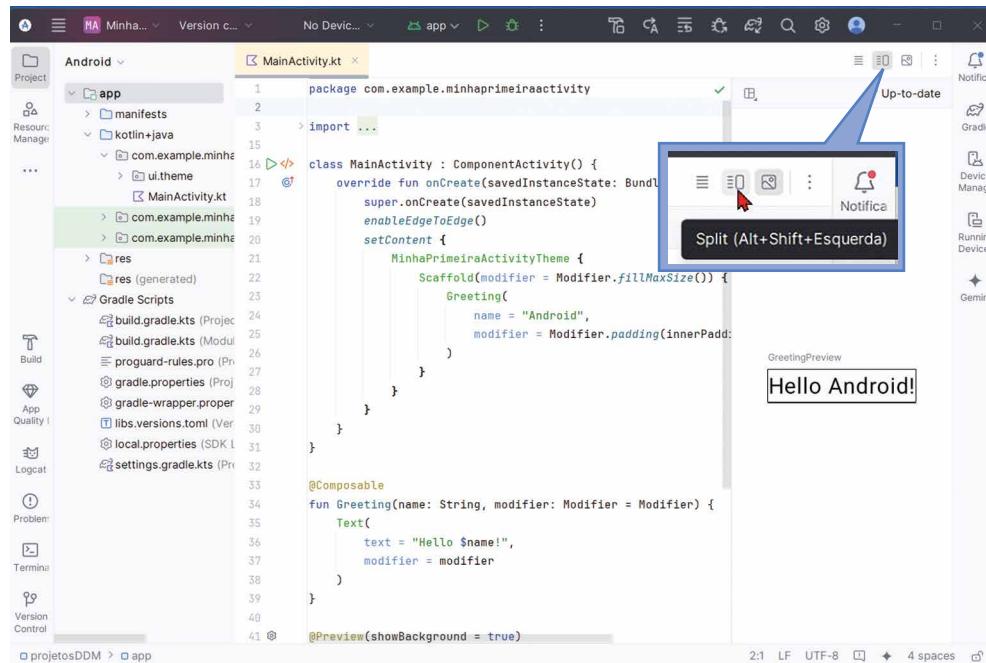


Figura 66 – Visualização padrão

Durante a programação, trabalhamos com três telas principais:

- A visualização do projeto exibe arquivos e pastas do seu projeto.
- A visualização do código é onde você edita o código.
- A visualização de design permite ver a aparência do aplicativo.

Unidade II

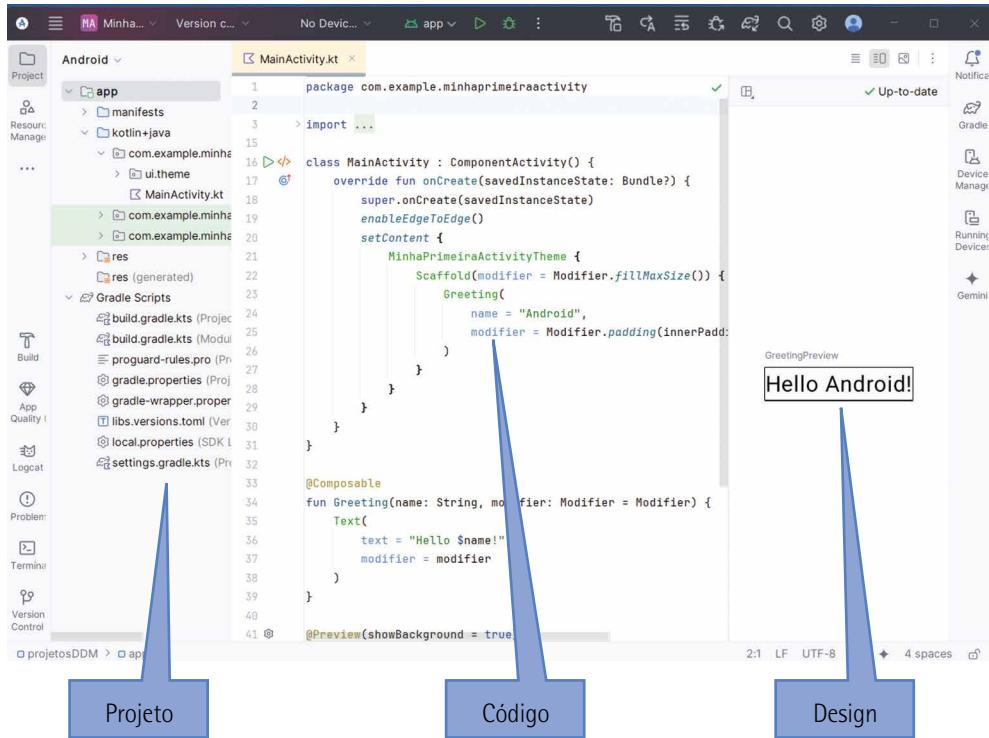


Figura 67 – Visualização padrão



Você pode recolher o painel de projeto clicando no botão Hide no canto superior direito do painel.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

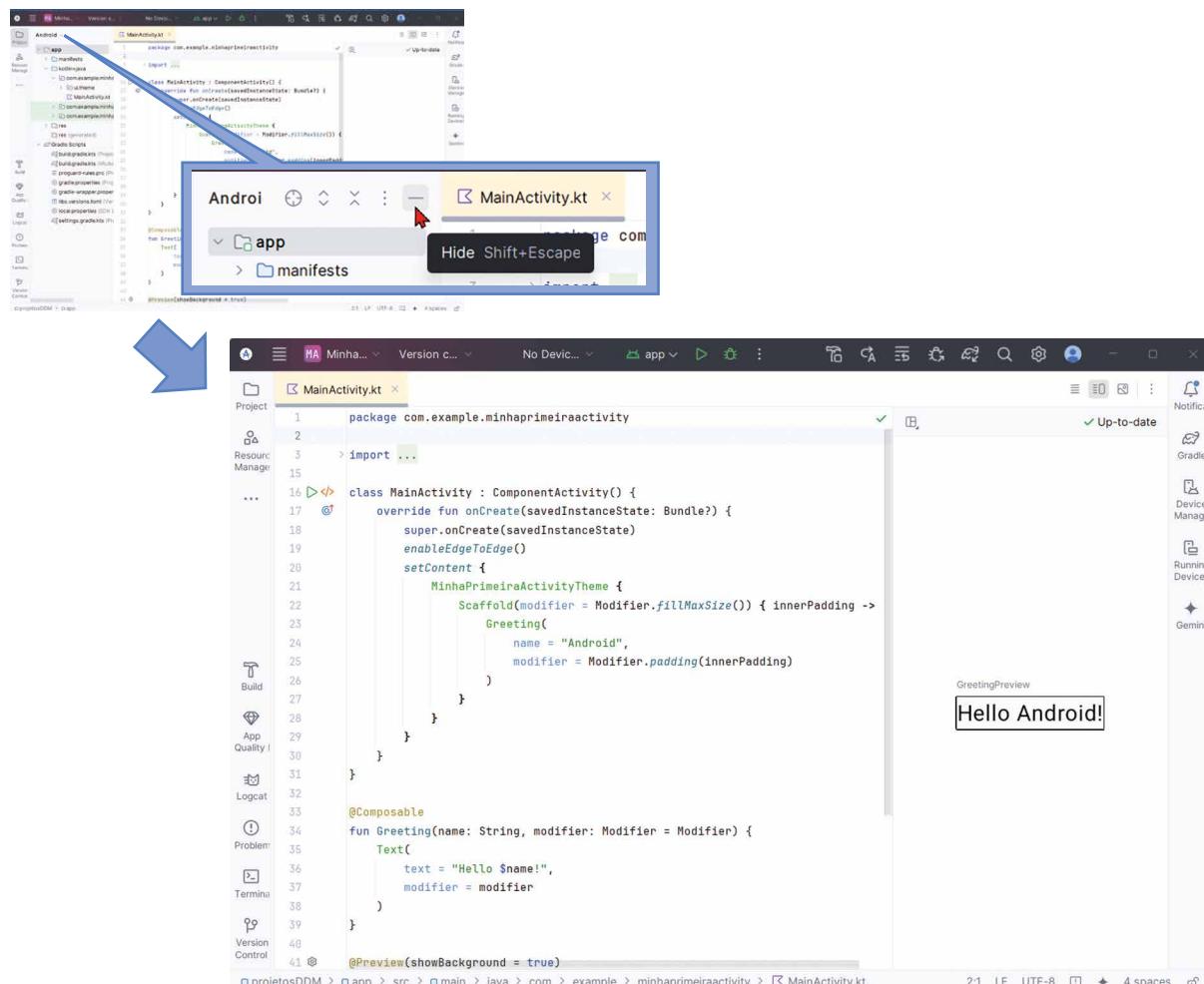


Figura 68

Para reabrir, clique em Project.

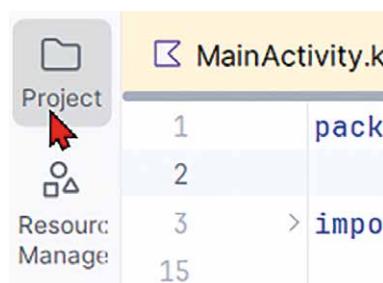


Figura 69

Para abrir ou fechar outros painéis, sempre procure o botão _.

5.3.1 Estrutura do projeto

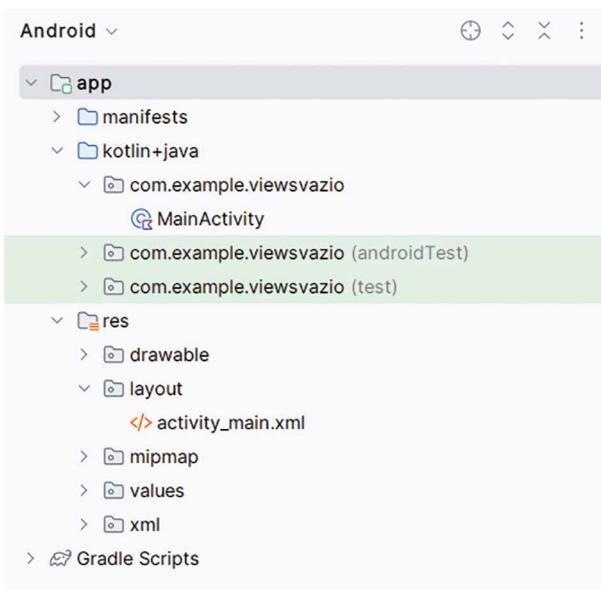


Figura 70 – Estrutura de pastas de um projeto

Por padrão, o Android Studio exibe os arquivos do projeto na visualização de projetos Android, conforme ilustrado na figura 70. Essa visualização é organizada por módulos, permitindo acesso rápido aos principais arquivos de origem do projeto. Todos os arquivos de build podem ser encontrados no nível superior em Gradle Scripts.

Cada módulo de aplicativo contém as seguintes pastas:

- **manifests**: contém o arquivo `AndroidManifest.xml`, um componente essencial de qualquer projeto Android. Ele fornece informações importantes sobre o aplicativo para as ferramentas de build do Android, o sistema operacional e a Google Play Store. No manifesto, declaram-se os componentes do aplicativo, como atividades, serviços, receptores de broadcast e provedores de conteúdo. Além disso, é onde se especificam as permissões necessárias para o aplicativo acessar partes protegidas do sistema ou de outros aplicativos e se definem as características de hardware e software que o aplicativo requer. Em resumo, o `AndroidManifest.xml` atua como um mapa que orienta o sistema operacional sobre como executar e gerenciar o aplicativo.
- **kotlin + java**: contêm os arquivos de código-fonte em Kotlin e Java, incluindo o código de `MainActivity`.
- **res**: contém todos os recursos não relacionados ao código, como strings de interface e imagens em bitmap. O Android Studio utiliza esses recursos para construir o aplicativo. Quando se referencia um recurso no seu código, o compilador do Android substitui essa referência pelo ID do recurso, permitindo que o sistema operacional carregue o recurso correto durante a execução.

do aplicativo. A pasta res é organizada em diversas subpastas, cada uma com uma finalidade específica: drawable. Armazena imagens e outros recursos gráficos.

- **layout**: contém os arquivos XML que definem os layouts da sua interface.
- **mipmap**: guarda os ícones do seu aplicativo.
- **values**: armazenam valores simples, como strings, cores, dimensões e estilos.
- **menu**: contém os arquivos XML que definem os menus do seu aplicativo.
- **anim**: armazena animações.
- **raw**: armazena arquivos brutos, como arquivos de áudio ou fontes.

5.4 Emulador

O emulador do Android Studio é uma ferramenta essencial para desenvolvedores Android, pois permite simular diversos dispositivos Android no computador. Funciona como um smartphone virtual, em que se testa os aplicativos antes de instalá-los em um dispositivo físico.

Com ele, é possível testar aplicativos em diferentes dispositivos e versões do Android, além de simular vários tamanhos de tela, resoluções, versões do sistema operacional e diferentes fabricantes de dispositivos. Isso garante que o aplicativo funcione de maneira correta em uma ampla variedade de dispositivos.

O emulador permite a depuração do aplicativo de forma fácil e rápida. Pode definir pontos de interrupção, inspecionar variáveis e acompanhar o fluxo de execução do seu código.

O Android Studio oferece diversas maneiras de emular um dispositivo Android para testar seus aplicativos. Cada método possui vantagens e é ideal para diferentes cenários. Nativamente, há duas opções: espelhar um dispositivo físico conectado via USB ou wi-fi se não utilizar o Android Virtual Device (AVD).

5.4.1 Espelhar um dispositivo físico

Muitas vezes, devido a limitações de hardware, como falta de memória RAM ou processadores desatualizados, pode-se utilizar um dispositivo externo como emulador. Isso ajuda a economizar o uso da memória e a CPU do computador. Essa técnica se chama espelhamento de dispositivo físico.

No Android Studio, o espelhamento pode ser feito conectando o dispositivo Android ao computador via cabo USB ou via wi-fi.

Espelhamento via wi-fi

No caso do espelhamento via wi-fi, tanto o computador quanto o dispositivo devem obrigatoriamente estar conectados à mesma rede wi-fi.

No seletor de dispositivos, escolha Pair Devices Using Wi-Fi. Em seguida aparecerá uma janela com um QR code. Escaneie e deixe instalar.

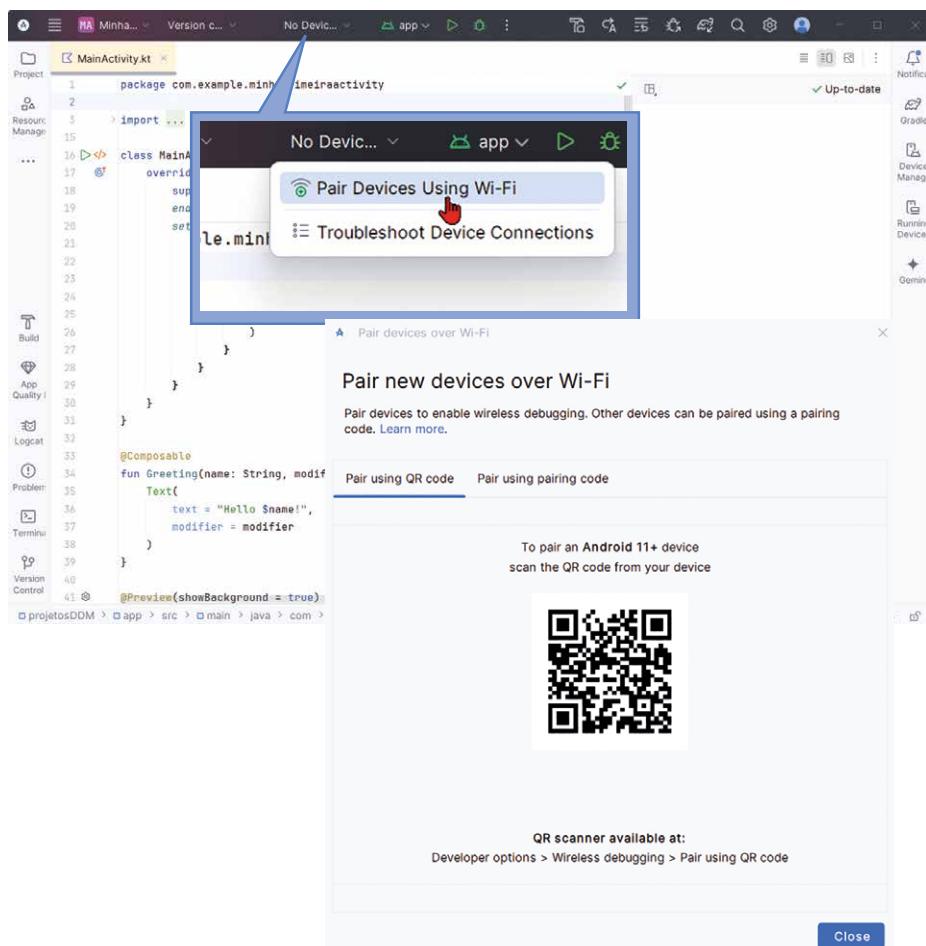


Figura 71 – Espelhamento via wi-fi

Espelhamento via cabo USB

Para espelhar via cabo USB, o computador deve ter um driver de conexão com o dispositivo. Na maioria dos casos, ele é instalado automaticamente na primeira conexão. Em alguns dispositivos específicos, pode ser necessário procurar o driver no site do fabricante. Outra coisa importante é a autorização para a depuração USB no dispositivo. Essa opção somente estará disponível se o dispositivo estiver no modo desenvolvedor, que está desativado por padrão. Para ativá-lo, siga normalmente estes passos:

- **Abra as configurações:** vá ao menu de configurações do dispositivo.
- **Sobre o telefone:** role até encontrar a opção "Sobre o telefone" ou "Sobre o dispositivo" e toque nela.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

- **Número da versão:** encontre a opção "Número da versão" ou "Número da compilação". Toque nela repetidamente (geralmente sete vezes) até ver uma mensagem informando que o modo desenvolvedor foi ativado.
- **Volte para configurações:** volte ao menu principal de configurações.
- **Busque:** faça uma busca na configuração para depurador USB.



O uso do modo desenvolvedor é técnico e nunca deve ser alterado, exceto sob orientação de especialistas.

Feitos esses passos, conecte o computador com o dispositivo e permita o uso de dados no celular. No computador espere o drive ser instalado e aguarde o sinal sonoro de conexão. Se a conexão estiver estabelecida, o seu dispositivo deverá estar disponível na lista de dispositivos.

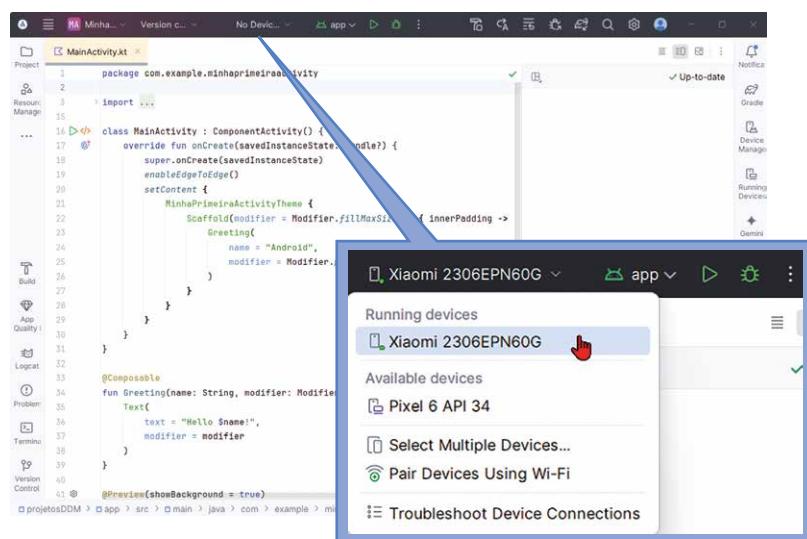


Figura 72 – Dispositivo físico conectado via cabo USB



Em caso de dúvida, consulte executar apps em um dispositivo de hardware.

DEVELOPERS. Executar apps em um dispositivo de hardware. [s.d.]e.
Disponível em: <https://shre.in/M6ZJ>. Acesso em: 6 mar. 2025.

Permita instalar no dispositivo.

5.4.2 Espelhar via AVD

O AVD é uma configuração que define as características de um emulador Android. Ao criar um AVD, há de se especificar o modelo de dispositivo, a versão do sistema operacional, a quantidade de RAM, o tamanho da tela e outras configurações relevantes. Isso permite simular diferentes dispositivos e testar seus aplicativos em diversos cenários. Para otimizar o desempenho do emulador, é recomendado que o computador atenda aos requisitos mínimos do sistema e que ajuste as configurações do AVD para reduzir o consumo de recursos, como desativando a aceleração de hardware ou diminuindo a resolução da tela quando não for necessário.

Aqui estão alguns pontos importantes sobre o AVD:

- **Criação e gerenciamento:** criar e gerenciar AVDs usando o Device Manager no Android Studio. Isso permite a configuração de dispositivos virtuais com diferentes perfis de hardware e versões do sistema operacional.
- **Perfis de hardware:** o AVD inclui um perfil de hardware que define as características do dispositivo, como tamanho da tela, resolução, memória e outros recursos. Você pode usar perfis predefinidos ou criar e personalizar seus próprios perfis.
- **Imagens de sistema:** cada AVD usa uma imagem de sistema que corresponde a uma versão específica do Android. Essas imagens podem incluir APIs do Google e outros componentes necessários para testar seu aplicativo.

Para instalar novos emuladores, acesse o Device Manager na barra lateral direita.

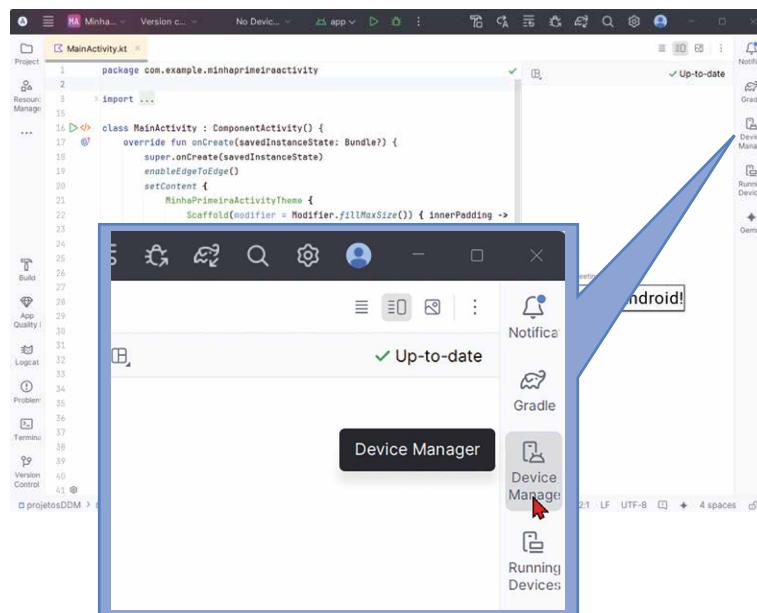


Figura 73 – Acesso ao Device Manager

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Na tela do Device Manager, clique no sinal de + para adicionar um dispositivo e escolha Create Virtual Device.

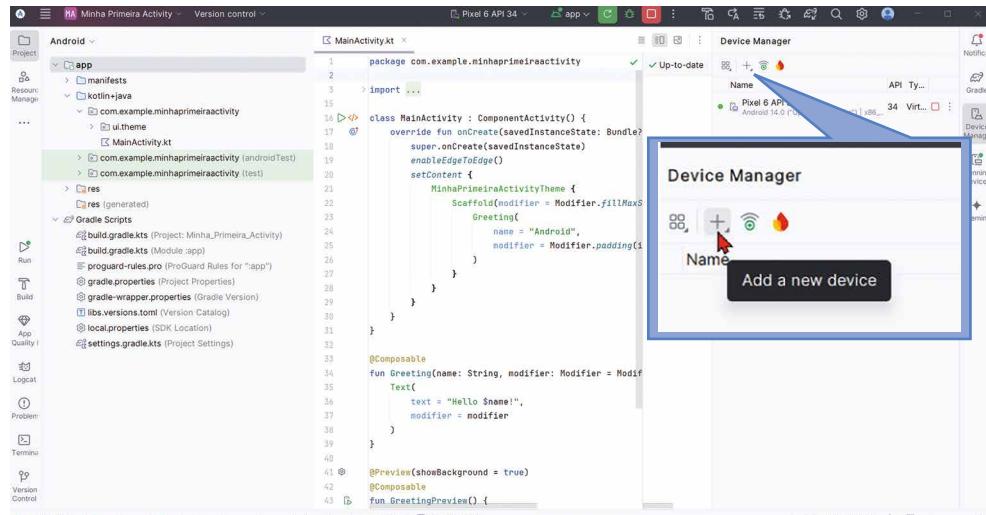


Figura 74 – Adicionar um dispositivo

A tela abrirá para a escolha do hardware (figura 75). Escolher modelos mais recentes ocupará mais memória e poderá incluir recursos que modelos mais antigos não tinham. Sem perceber, você pode utilizar esses recursos, desenvolvendo aplicativos que apenas funcionam em dispositivos com hardware novo. Por outro lado, optar por equipamentos antigos pode ocupar menos memória, mas eles talvez não suportem sistemas operacionais mais atualizados. Adotaremos o modelo Pixel 6 do Google.

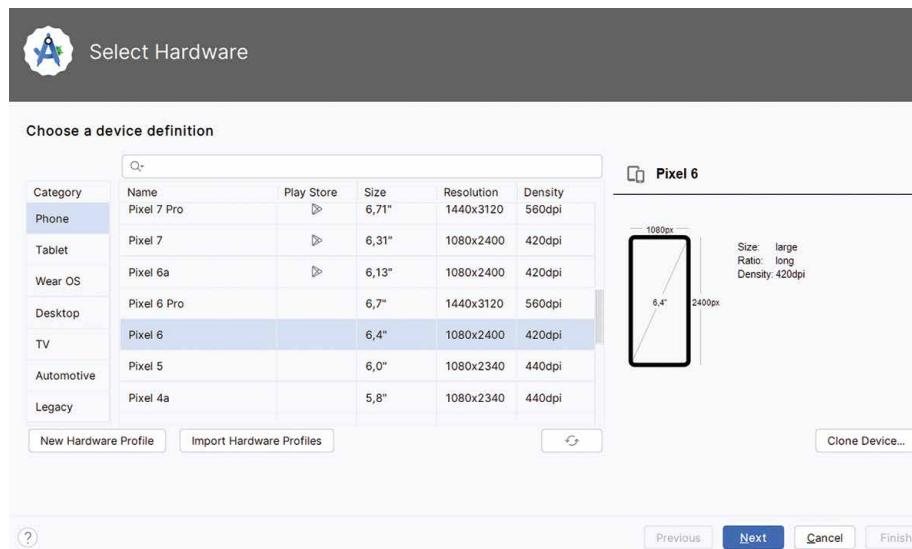


Figura 75 – Escolha do hardware

A próxima tela é a da escolha da versão do Android. A versão do sistema operacional segue também os critérios da escolha do hardware, uma atual sem necessariamente ser a última. Utilizaremos o UpsideDownCake (figura 76)..

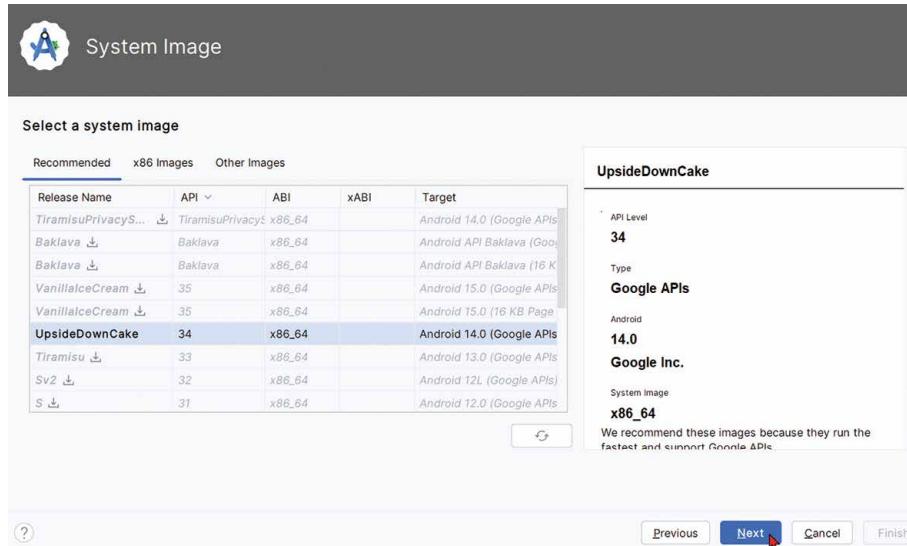


Figura 76 – Escolha do sistema operacional

O passo seguinte é confirmar e terminar a instalação.

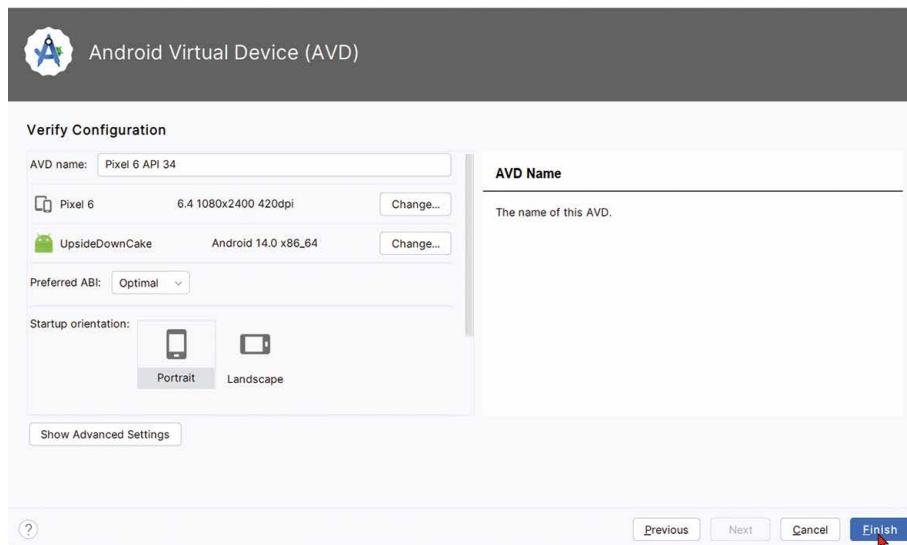


Figura 77 – Finalização da adição de um emulador

5.4.3 Executar via AVD



Figura 78

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Feche o gerenciador de dispositivos (Device Manager), clicando no sinal de _ . Para executar, clique em Run app no menu superior e espere até todo o Gradle ser montado.

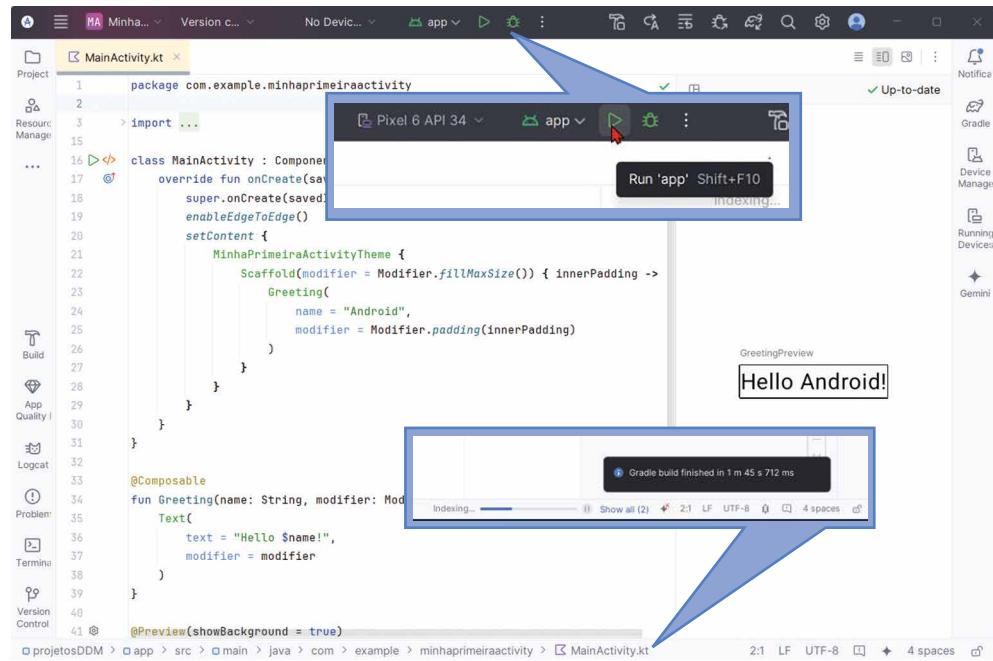


Figura 79 – Executando o emulador

Uma vez montado o emulador, ela ficará o editor.

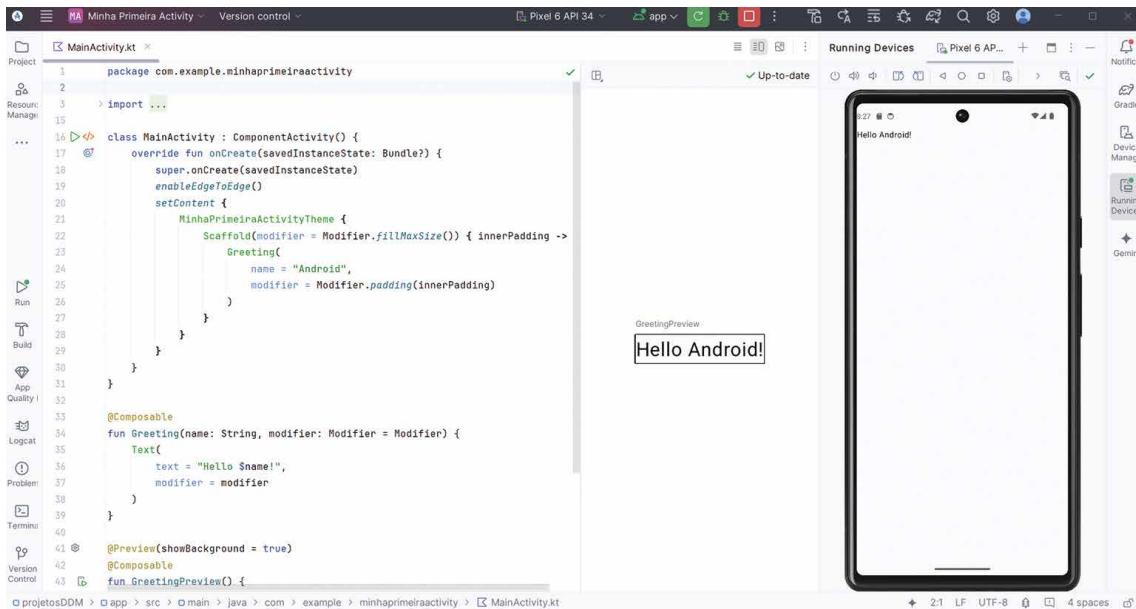


Figura 80 – Emulador ativo



Conforme o processador, durante a montagem do dispositivo virtual, o sistema pode perder a conexão com o computador. Isso é normal, clique em wait. Caso a seta verde do run app retorne sem o emulador estar funcionando, clique novamente, pois a montagem continuará a partir da ponte interrompida.

6 INTERFACES GRÁFICAS

Interface XML (views)

As views são os elementos visuais em aplicativos Android. Elas representam os componentes da interface com os quais o usuário interage diretamente. No Android Studio, podemos criar e manipular views tanto visualmente, com um editor gráfico (arquivos XML), quanto programaticamente, utilizando linguagens como Java ou Kotlin. Essa dualidade permite construir interfaces complexas e personalizadas, combinando rapidez e praticidade do design visual da programação.

6.1 Iniciando o projeto com views

Para iniciar um novo projeto de views no Android Studio, inicie em New Project e selecione o template Empty Views Activity para um projeto básico com uma activity principal.

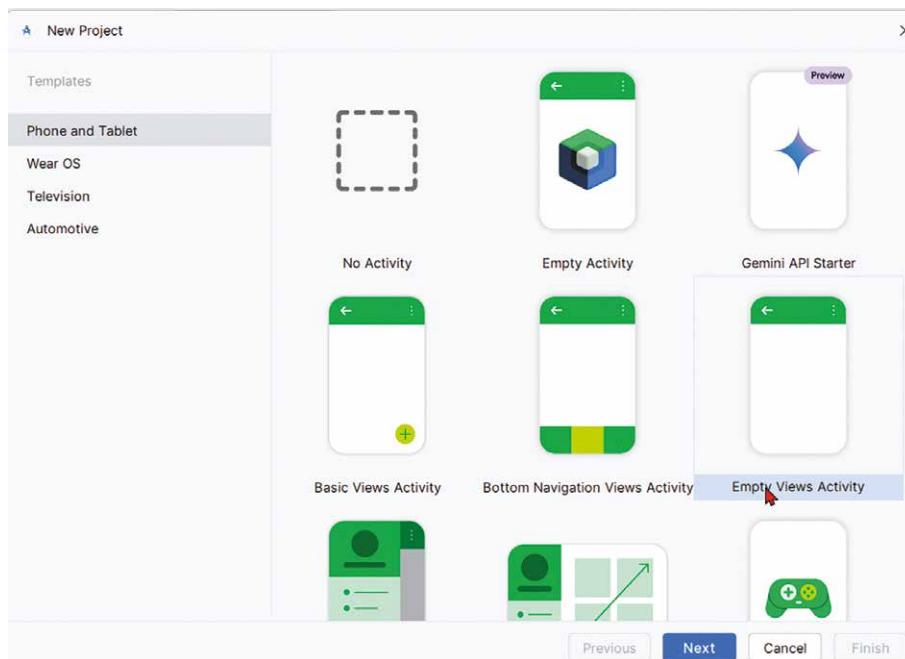


Figura 81 – Opção para iniciar um novo projeto views

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Assim como no início do primeiro projeto, preenchemos a tela de propriedades iniciais da nova activity vazia, alterando o nome para Minha Activity Views, deixando os outros campos inalterados e finalizando ao clicar em Finish.

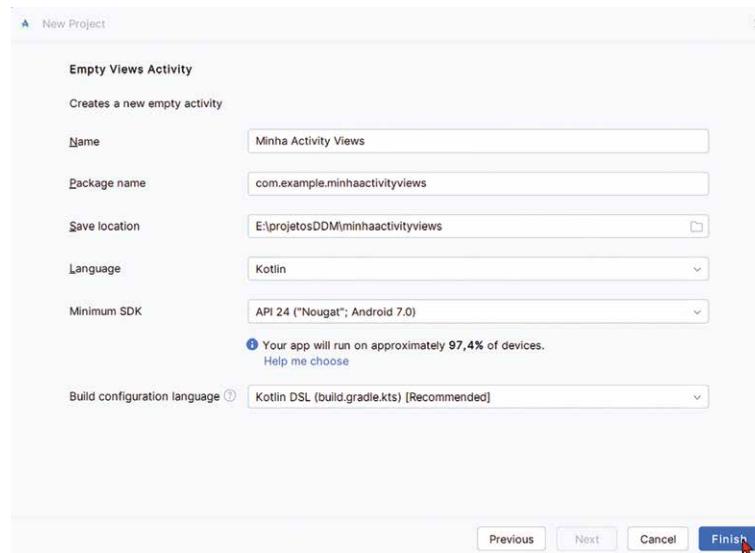


Figura 82 – Configuração inicial da nova activity

Espere o Gradle terminar a sincronização.

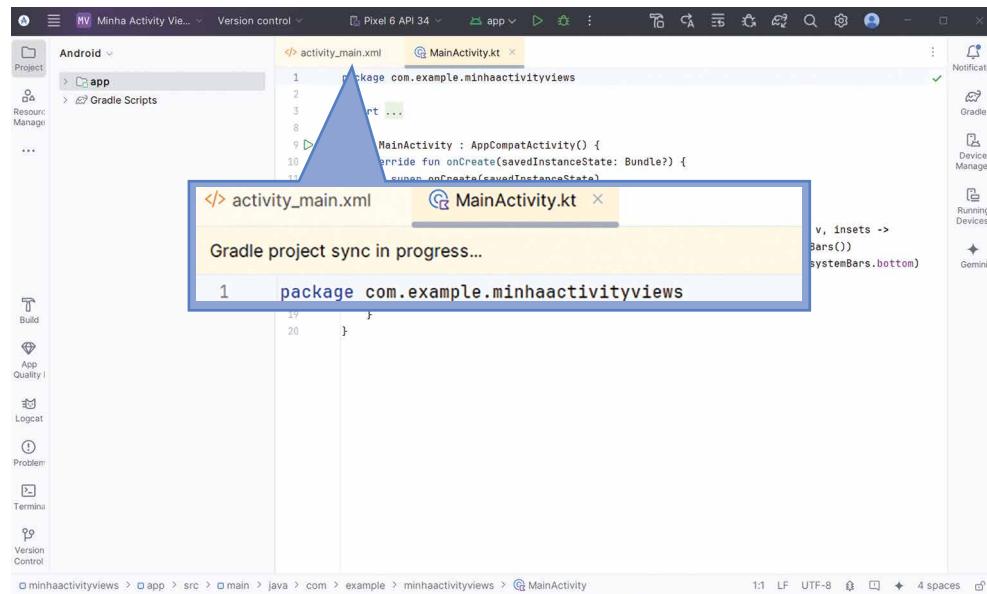


Figura 83 – Status do gradle em construção

6.1.1 Gradle

O código-fonte, escrito em uma linguagem como Java ou Kotlin, é um conjunto de instruções que o computador não entende diretamente. O processo de build, gerenciado pelo Gradle no Android Studio, compila esse código, transformando-o em um formato executável (APK) que o dispositivo Android pode interpretar e executar.

No Android Studio, o Gradle é uma ferramenta de automação de build usada para compilar, testar e empacotar aplicativos Android. Ele permite gerenciar dependências, configurar diferentes variantes de build e automatizar várias tarefas do processo de desenvolvimento. O Gradle funciona como um mestre de obras, coordenando todas as tarefas necessárias para transformar o código em um aplicativo Android executável.

A cada inicialização ou atualização de biblioteca, o Gradle precisa ser sincronizado para montar corretamente o aplicativo. Sempre espere a sincronização do Gradle terminar antes de prosseguir.



Saiba mais

Para informações completas sobre o processo de build, consulte a visão geral do build do Gradle, acessando:

DEVELOPERS. *Visão geral do build do Gradle*. [s.d.]k. Disponível em: <https://shre.ink/M6FS>. Acesso em: 6 mar. 2025.

6.2 Interface

Note que, ao carregar o projeto, duas abas aparecem na janela de código: a primeira é activity_main.xml e a segunda é MainActivity.kt (figura 84).

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

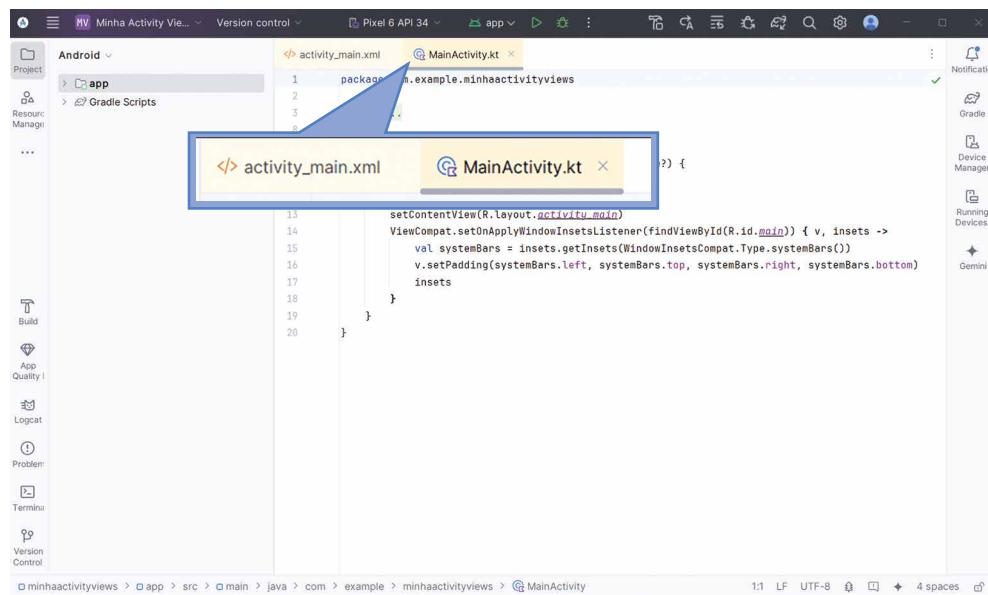


Figura 84 – As duas janelas principais da janela de código do views

6.2.1 MainActivity.kt

O MainActivity.kt é um arquivo fundamental em qualquer projeto Android criado utilizando a linguagem Kotlin. Ele representa a atividade principal do aplicativo, ou seja, a primeira tela que o usuário verá quando abrir o app. Ele é o ponto central de controle do aplicativo Android, conectando a interface do usuário à lógica do seu código. Para criar aplicativos é fundamental entender como ele funciona e como se relaciona com as views.

O arquivo MainActivity.kt está localizado dentro do diretório do projeto no Android Studio.

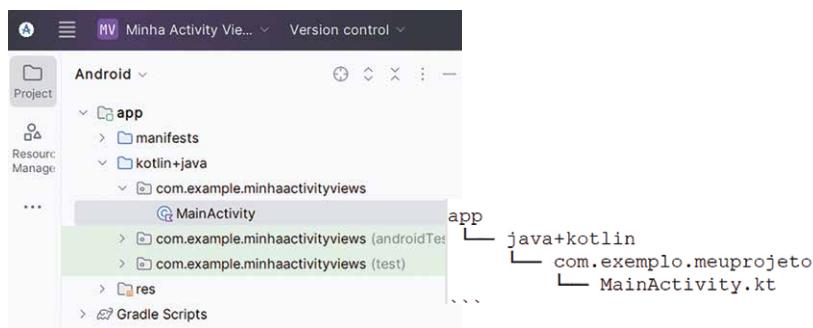


Figura 85 – Localização do arquivo MainActivity.kt

Se a tela do editor for fechada acidentalmente, basta dar um duplo clique em MainActivity na árvore do projeto para reabri-la.



```
activity_main.xml MainActivity.kt
1 package com.example[minhaactivityviews]
2
3 > import ...
4
5 <></> class MainActivity : AppCompatActivity() {
6     @Override fun onCreate(savedInstanceState: Bundle?) {
7         super.onCreate(savedInstanceState)
8         enableEdgeToEdge()
9         setContentView(R.layout.activity_main)
10        ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main)) { v, insets ->
11            val systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars())
12            v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom)
13            insets
14        }
15    }
16 }
17
18 }
```

Figura 86 – Código inicial do MainActivity.kt

Embora o template escolhido se chame Empty Views Activity (Activity Views Vazio), ele vem com um código funcional inicial que verifica se tudo funciona corretamente no Android Studio. Esse aplicativo exibe Hello World! no centro da tela.

```
package com.example[minhaactivityviews]
```

A linha package no arquivo MainActivity.kt especifica o pacote ao qual o arquivo pertence. Em Kotlin (e Java), um pacote é uma forma de organizar classes e arquivos de forma hierárquica, semelhante a diretórios em um sistema de arquivos.

```
import...
```

A diretiva import no arquivo MainActivity.kt serve para importar classes e pacotes de outras partes do código, como a biblioteca padrão do Android ou bibliotecas externas. Essa importação torna as funcionalidades dessas classes e pacotes acessíveis dentro do seu código, permitindo utilizar métodos, propriedades e outros recursos oferecidos por essas bibliotecas sem precisar reimplementá-los.

```
class MainActivity : AppCompatActivity() {  
    // código da classe  
}
```

Figura 87

Declara a classe MainActivity que herda de AppCompatActivity. Isso significa que MainActivity é uma atividade que pode usar todos os recursos de compatibilidade fornecidos pela AppCompatActivity. A classe é o ponto de início do aplicativo.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        Activity a ser executada  
    }  
}
```

Figura 88

Sobrescreve-se o método `onCreate` da `AppCompatActivity`. Esse método é chamado quando a atividade é criada. O parâmetro `savedInstanceState` contém o estado anterior da atividade, se ela foi encerrada e recriada.

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        enableEdgeToEdge()  
        setContentView(R.layout.activity_main)  
        ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main)) { v, insets ->  
            val systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars())  
            v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom)  
            insets  
        }  
    }  
}
```

Figura 89

Chama-se o método `onCreate` da superclasse (`AppCompatActivity`) para executar a inicialização padrão da atividade.

Utiliza um método personalizado `enableEdgeToEdge()`. Esse método provavelmente configura a interface do usuário para usar o modo de tela cheia, em que o conteúdo pode se estender até as bordas da tela, incluindo as áreas de status e navegação.

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        enableEdgeToEdge()  
        setContentView(R.layout.activity_main)  
        ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main)) { v, insets ->  
            val systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars())  
            v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom)  
            insets  
        }  
    }  
}
```

Figura 90

A função `enableEdgeToEdge()` declara automaticamente que o app precisa ser disposto de borda a borda e ajusta as cores das barras do sistema. Ela deixa as barras do sistema transparentes, exceto no modo de navegação com três botões, em que a barra de status recebe uma tela de fundo translúcida. As cores dos ícones do sistema e do filtro são ajustadas com base no tema claro ou escuro do sistema.

Unidade II

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContentView(R.layout.activity_main)
        ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main)) { v, insets ->
            val systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars())
            v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom)
            insets
        }
    }
}
```

Figura 91

Define o layout da atividade para o arquivo XML activity_main. Isso infla o layout XML e o exibe na tela. O método setContentView, fornecido pela classe AppCompatActivity, recebe como parâmetro o ID do recurso (geralmente definido em R.java) do arquivo de layout que se deseja inflar (exibir) como o conteúdo da atividade. Nesse caso, é activity_main.xml.

Esta é a linha fundamental para o funcionamento do aplicativo. Com apenas o setContentView e o onCreate já permite fazer o build.

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContentView(R.layout.activity_main)
        ViewCompat.setOnApplyWindowInsetsListener(findViewById(R.id.main)) {
            v, insets ->
            val systemBars = insets.getInsets(WindowInsetsCompat.Type.systemBars())
            v.setPadding(systemBars.left, systemBars.top, systemBars.right, systemBars.bottom)
            insets
        }
    }
}
```

Figura 92

Esse bloco de código define um listener para os insets da janela. Com o objetivo de explicar melhor, imagine que a tela do seu celular é um quadro. Dentro desse quadro, há a área principal do seu aplicativo (o que vê na maior parte do tempo). No entanto, algumas partes do quadro são reservadas para o sistema operacional, como a barra de status (onde aparecem a hora e o sinal) e a barra de navegação (com os botões Voltar, Home e Recentes).

Essas áreas ocupam espaço na tela e são chamadas de insets. Nesse caso, um listener de insets funciona como um vigia que monitora essas áreas reservadas. Ele observa se o tamanho ou a posição dessas áreas mudam (por exemplo, quando você esconde a barra de status para ter mais espaço na tela). Quando isso acontece, o listener notifica o aplicativo, permitindo ajustar a interface para que tudo continue funcionando corretamente e com uma boa aparência.

O listener de insets é uma ferramenta importante para garantir que o aplicativo se adapte de forma flexível a diferentes dispositivos e configurações, oferecendo uma interface consistente ao usuário.

6.2.2 activity_main.xml

O arquivo `activity_main.xml` é um layout utilizado em aplicativos Android. Ele define a interface do usuário para a atividade principal do aplicativo. Escrito em XML, esse arquivo inclui elementos como botões, textos, imagens, entre outros.

Cada elemento no XML é traduzido em classes Kotlin ou Java em tempo de execução. Ele estabelece a estrutura visual da tela inicial, organizando e apresentando os elementos (botões, textos, imagens etc.) ao usuário. Os elementos definidos no XML são instanciados no programa Kotlin para serem utilizados de modo dinâmico.



Figura 93 – Elementos do `activity_main.xml` sendo instanciados no `MainActivity.kt`

O arquivo `activity_main.xml` está localizado na pasta `layout` dentro dos recursos do projeto.

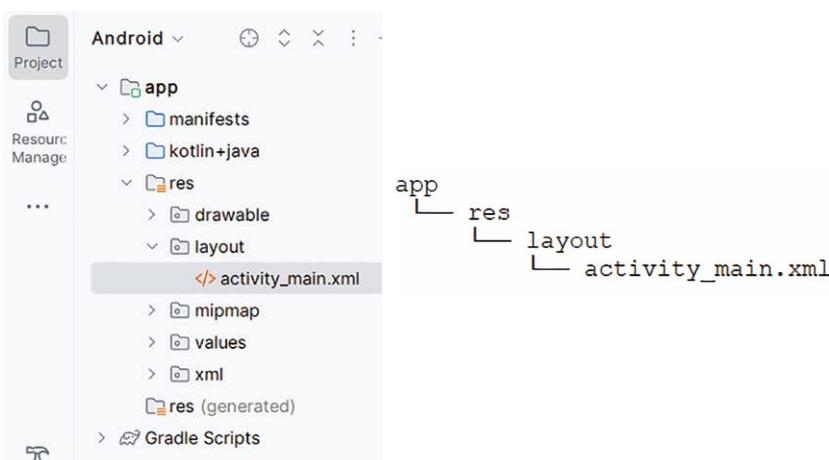


Figura 94 – Localização do arquivo `activity_main.xml`

Observação

A pasta res no Android Studio é onde ficam armazenados todos os recursos (resources) do aplicativo. Esses recursos incluem layouts, imagens, strings, animações, cores, menus, entre outros. A organização dos recursos na pasta res facilita a manutenção e a internacionalização do aplicativo.

- **Drawable:** imagens e gráficos, como bitmaps e arquivos XML que definem formas e animações.
- **Layout:** arquivos XML que definem a interface do usuário (UI) do aplicativo.
- **Values:** arquivos XML que definem valores como strings, cores, dimensões e estilos.
- **Menu:** arquivos XML que definem menus de opções e de contexto.
- **Anim:** arquivos XML que definem animações de interpolação (tween animations).
- **Raw:** arquivos em seu formato bruto, como, por exemplo, arquivos de áudio ou vídeo.

Ela é referenciada como R, por isso, ao escrever R.layout.activity_main, o sistema já reconhece como app\res\layout\activity_main.xml.

Para editar o xml, clique na aba superior que é aberta automaticamente quando o programa é criado (figura 95).

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

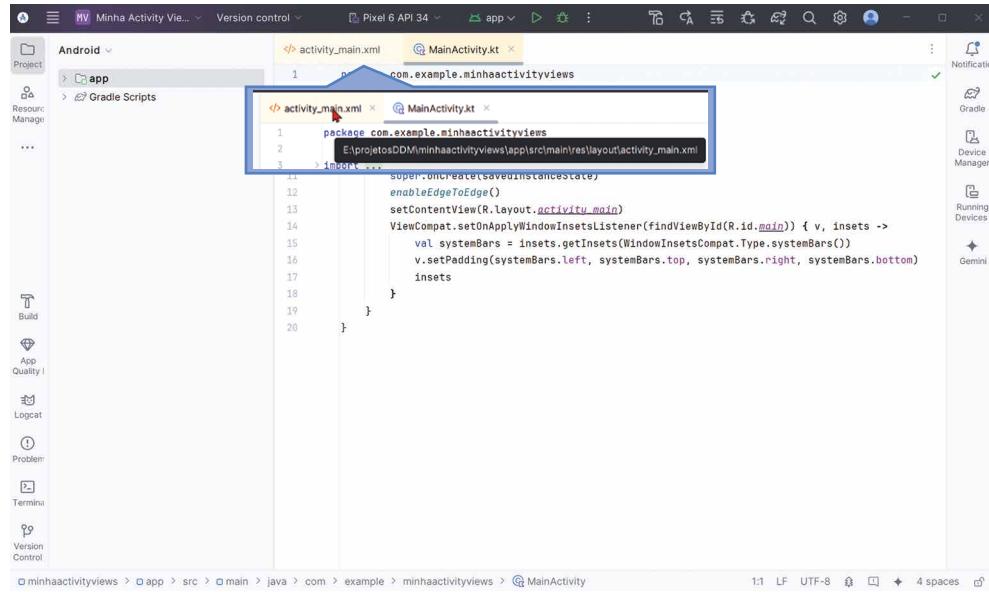


Figura 95

A edição pode ser exibida de três modos diferentes, que podem ser selecionados no canto superior direito da janela do xml: code, design e split.

- **Code:** exibe apenas o código XML. É ideal para desenvolvedores que preferem escrever e editar o XML manualmente. Pode ver e modificar diretamente a estrutura do layout, adicionar atributos e ajustar propriedades.



Figura 96

- **Design:** oferece uma interface visual em que se pode arrastar e soltar componentes para criar o layout. É útil para visualizar rapidamente como o layout ficará em diferentes dispositivos e orientações. O design editor também permite ajustar propriedades dos componentes pelos painéis de propriedades.

Unidade II

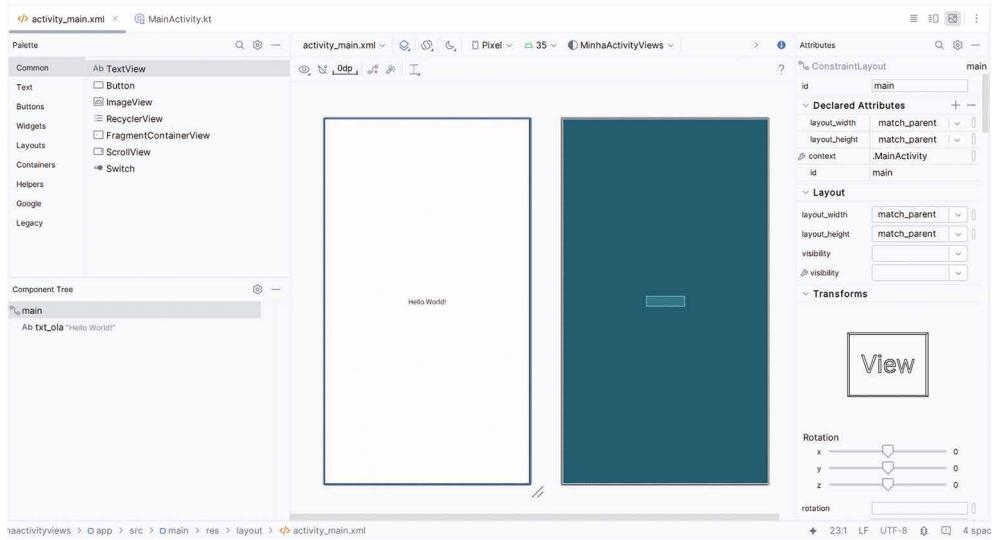


Figura 97

- **Split:** combina os dois anteriores, mostrando o código XML e a visualização do design lado a lado. Isso permite analisar as mudanças em tempo real enquanto edita o XML, proporcionando uma maneira eficiente de ajustar e refinar seu layout1.

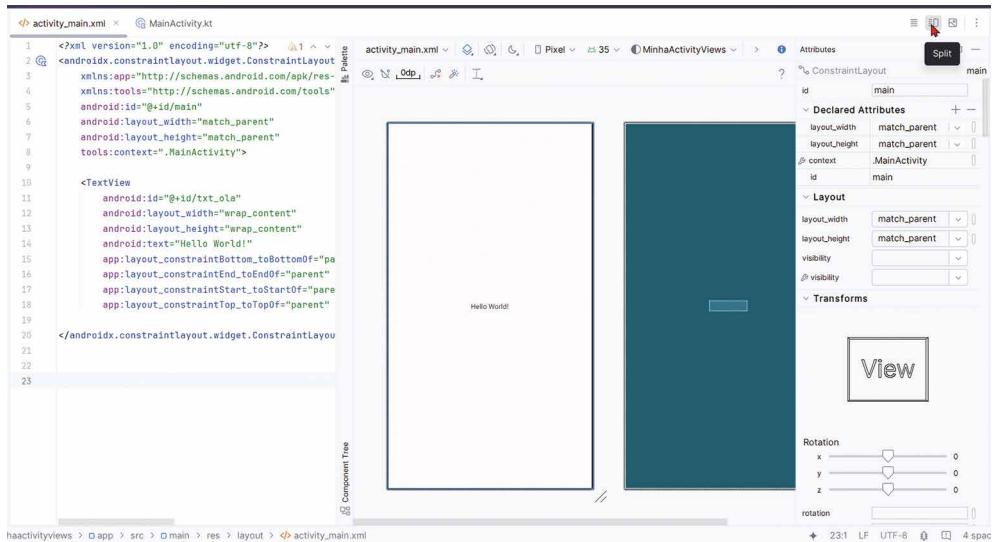


Figura 98

Clicando o botão de projetos no menu esquerdo, abre ou fecha a tela de projetos. Desta forma há a tela de edição mostrando os painéis da árvore de projeto, do código e do design.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

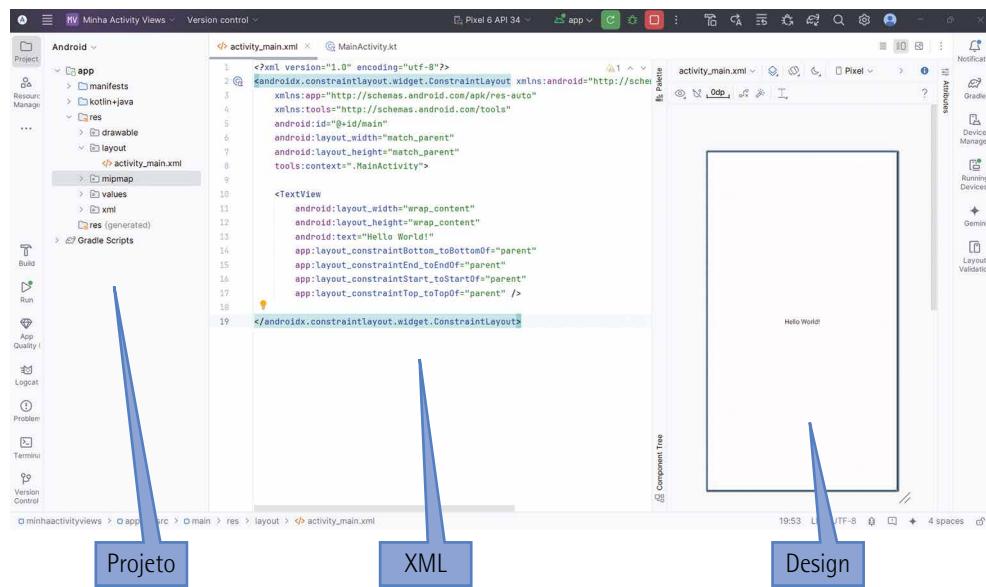


Figura 99 – Janela de edição no modo split, com o painel de projeto

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res-auto"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Figura 100 – Código inicial do activity_main.xml

Com o template escolhido, Empty Views Activity (Activity Views Vazio), ele vem com um código xml para exibir Hello World! no centro da tela. Analisemos o código. A primeira linha declara a versão do XML e o conjunto de caracteres usados.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Figura 101

A tag mais externa cerca todo o conteúdo da tela e define como será feita a administração dos elementos na tela ou o layout. Aqui, especifica-se que o layout usará o ConstraintLayout da biblioteca androidx. É um gerenciador de layout avançado que permite posicionar elementos de forma flexível, utilizando restrições (constraints). Isso significa que pode definir como os elementos devem se alinhar e se relacionar uns com os outros, proporcionando um controle preciso sobre o design da interface do usuário.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    Conteúdo do Constraint Layout

</androidx.constraintlayout.widget.ConstraintLayout>
```

Figura 102

Os elementos mais internos são alocados dentro do layout. Eles podem ser elementos concretos como botões, textos, imagens, até outros layouts redefinido a disposição específica de áreas. No exemplo tem-se um elemento TextView, que exibirá texto na tela.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"           World!">
    </TextView>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Figura 103

Além da edição com código, altera-se o layout trabalhando diretamente com elementos na tela de design. Os elementos estão agrupados em um menu que normalmente fica recolhido no lado esquerdo da tela de design. Para mostrar, clique em palette na garagem direita da tela de design.

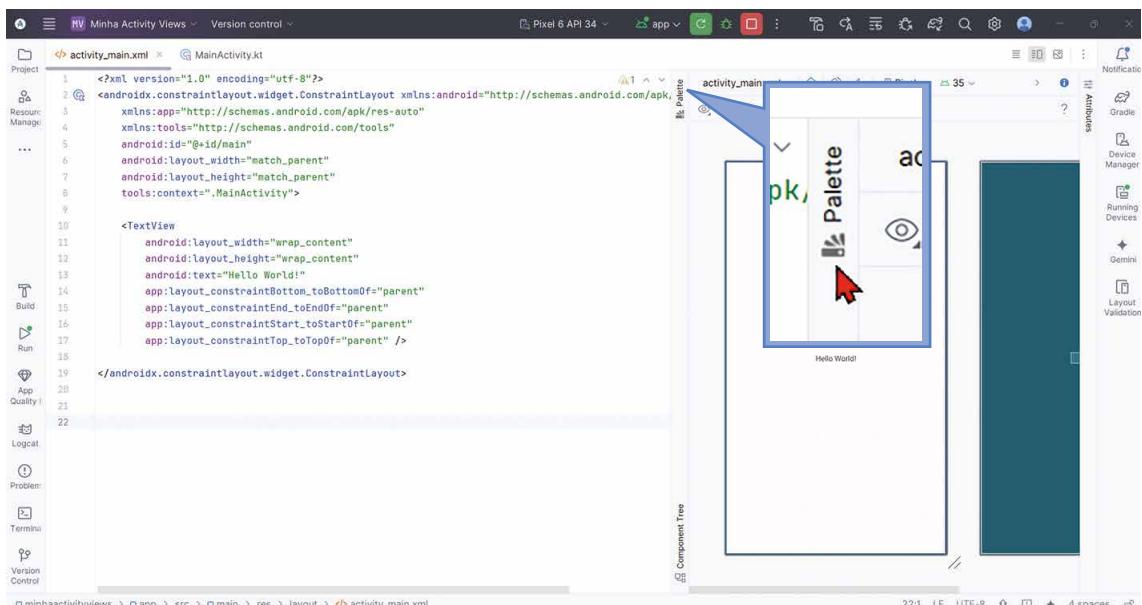


Figura 104 – Abrindo a paleta

A paleta permite arrastar e soltar componentes de interface, como botões, textos e layouts, diretamente na tela de design. Isso torna o processo de criação de layouts mais intuitivo.

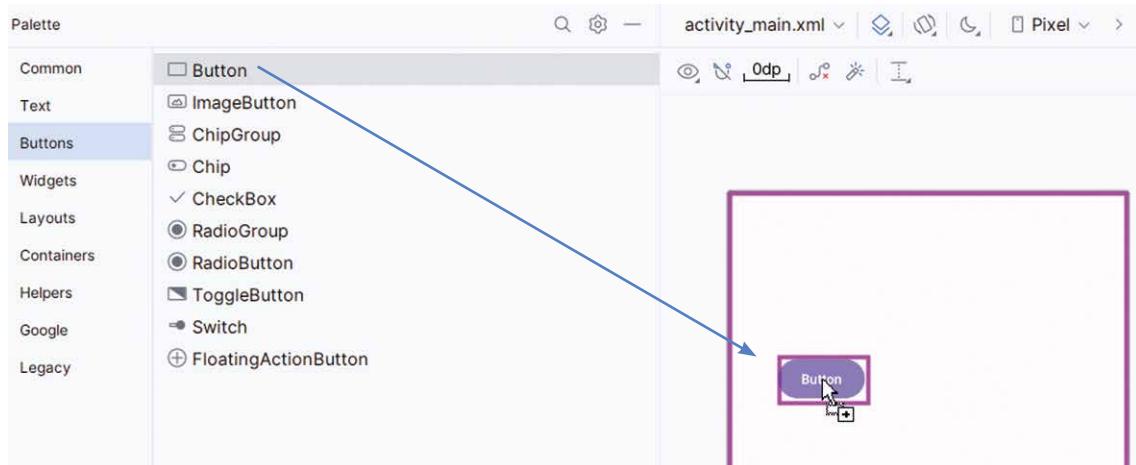


Figura 105

Quando se abre a paleta, os elementos estão organizados por funcionalidade, permitindo que os encontre e posicione na tela sem precisar saber exatamente o nome de cada um. Os principais grupos são:

- **Common:** contém os componentes mais frequentemente usados, como TextView, Button, ImageView e EditText.
- **Text:** inclui todos os componentes relacionados a texto, como TextView, EditText, Button, CheckBox, RadioButton e Switch.
- **Buttons:** agrupam diferentes tipos de botões, como Button, ImageButton, ToggleButton e FloatingActionButton.
- **Widgets:** encontram-se os elementos básicos da interface, como TextView (texto), Button (botões), ImageView (imagens), EditText (campo de texto para entrada de dados), além de ProgressBar, SeekBar, RatingBar e Spinner.
- **Layouts:** contêm os diferentes tipos de layouts que podem ser usados para organizar os elementos na tela, como LinearLayout, RelativeLayout, ConstraintLayout etc. Cada layout possui suas próprias características e é ideal para diferentes cenários.
- **Contêineres:** elementos que agrupam outros elementos, permitindo criar estruturas mais complexas. Um exemplo comum é o ViewGroup, que pode conter outros elementos como TextView e Button. Há também componentes que podem conter outros elementos, como ScrollView, HorizontalScrollView, ViewPager e RecyclerView.

Ao colocar um botão na tela, automaticamente o código é atualizado com uma tag referente ao botão colocado (figura 106).

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
</> activity_main.xml × MainActivity.kt  
1  <?xml version="1.0" encoding="utf-①1▲2▲1^>  
2  <androidx.constraintlayout.widget.ConstraintLayout  
3      xmlns:app="http://schemas.android.com/apk/res  
4      xmlns:tools="http://schemas.android.com/tools  
5      android:id="@+id/main"  
6      android:layout_width="match_parent"  
7      android:layout_height="match_parent"  
8      tools:context=".MainActivity">  
9  
10     <TextView  
11         android:layout_width="wrap_content"  
12         android:layout_height="wrap_content"  
13         android:text="Hello World!"  
14         app:layout_constraintBottom_toBottomOf="parent"  
15         app:layout_constraintEnd_toEndOf="parent"  
16         app:layout_constraintStart_toStartOf="parent"  
17         app:layout_constraintTop_toTopOf="parent"  
18  
19     <Button  
20         android:id="@+id/button"  
21         android:layout_width="wrap_content"  
22         android:layout_height="wrap_content"  
23         android:text="Button"  
24         tools:layout_editor_absoluteX="168dp"  
25         tools:layout_editor_absoluteY="419dp" />  
26  
27     </androidx.constraintlayout.widget.ConstraintLayout  
28
```

Figura 106

A tag Button também está sublinhada em vermelho, indicando a presença de um erro. Esse erro é comum ao inserir um novo elemento em um Constraint Layout, pois esse layout exige que os elementos sejam ancorados a outros widgets ou às bordas do layout. Essa ancoragem é feita por meio de atributos do elemento.

Essas ancoragens são definidas por atributos como app:layout_constraintTop_toTopOf, app:layout_constraintStart_toStartOf, entre outros. Para adicionar essas ancoragens, digitam-se os atributos diretamente no código XML ou utiliza-se o editor visual de atributos do Android Studio, que permite criar as ancoragens de forma intuitiva, arrastando e soltando linhas entre os elementos. Para abrir o editor de atributos na tela de design, basta clicar em Attributes.

Unidade II

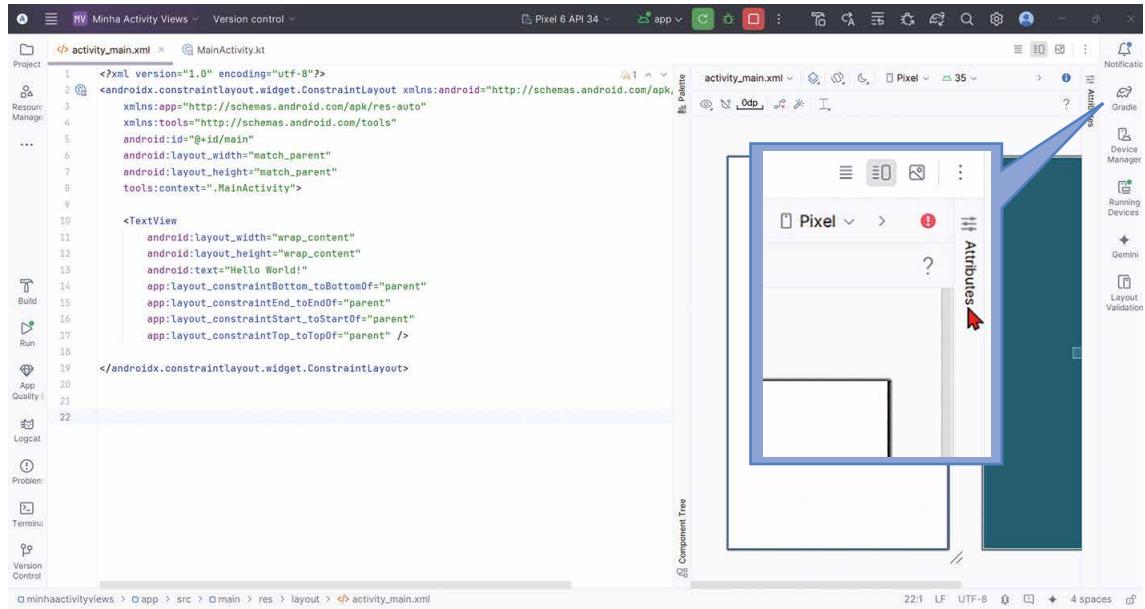


Figura 107 – Abrir o editor de atributos

Ao clicar em Attributes, a janela de atributos é aberta.

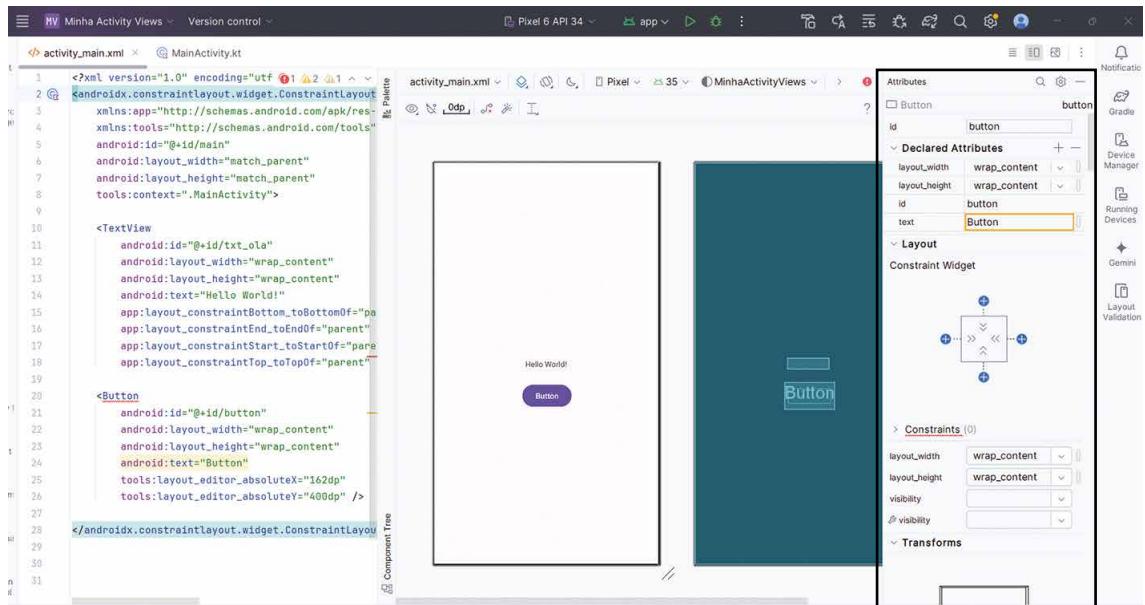


Figura 108 – Janela de edição dos atributos

Na tela de edição do código xml, o atributo id, identificação do elemento, é `@+id/button`:

```
<Button
    android:id="@+id/button"
```

Figura 109

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Clique no botão criado para alterar o foco para ele, o atributo id pode ser alterado, por exemplo, para btnOK.



Figura 110

Ao alterar e refatorar, observe novamente a tela do código xml. O atributo será alterado para btnOK.



Figura 111

Para corrigir o erro de falta de ancoragem (constraint), localize o Constraint Widget dentro de layout no editor de atributos e ancore no elemento anterior (TextView) e na lateral esquerda (parent:ConstraintLayout).

Primeiro clique no sinal + superior e crie o constraint (Create Top Constraint).

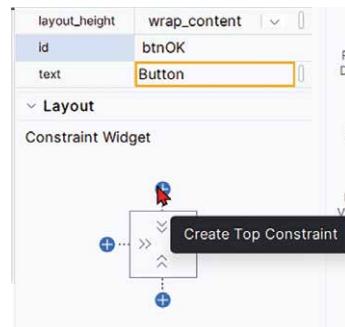


Figura 112

Depois clique no sinal de + da esquerda e crie o constraint da esquerda.

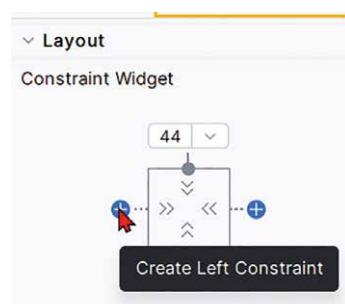


Figura 113

Observe que agora aparecem as setas indicando as ancoragens.

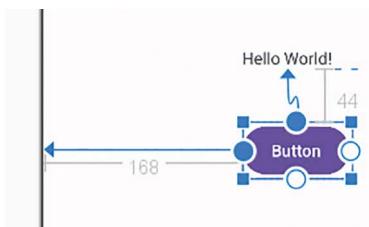


Figura 114

No código xml, o sublinhado vermelho foi desativado e dois novos atributos foram criados: `layout_constraintStart_toStartOf="parent"`, entre o início do layout-pai e o início do elemento, e o atributo `layout_constraintTop_toBottomOf="@+id/textView"`, ligando o topo do elemento à parte de baixo do `textView`.

```
<Button  
    android:id="@+id/btnOK"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:layout_marginStart="168dp"  
    android:layout_marginTop="44dp"  
    android:text="Button"  
    app:layout_constraintStart_toStartOf="parent"  
    app:layout_constraintTop_toBottomOf="@+id/textView" />
```

Figura 115

6.3 Views

A classe `view` é o ponto de partida para criar a interface de um aplicativo Android. Ela é o pai de uma vasta hierarquia de controles visuais (widgets) representando o elemento visual mais básico. Os widgets, como botões e caixas de texto, são tipos especializados de `views` que oferecem funcionalidades predefinidas. Layouts, como `LinearLayout` e `RelativeLayout`, são tipos de `ViewGroup` que organizam outras `views` na tela. Essa hierarquia de classes permite criar interfaces complexas e personalizadas.

A classe `view` serve como ponto de partida para a criação de todos os elementos visuais em aplicativos Android, formando a base de uma extensa hierarquia de classes. Todos os elementos visuais da interface, desde os mais simples até os mais complexos, estendem a classe `view`. Isso significa que uma `view` é o ancestral comum de todos os widgets, como botões e caixas de texto. Essa estrutura hierárquica permite uma organização clara e eficiente do código, facilitando a criação e a manutenção de interfaces de usuário.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

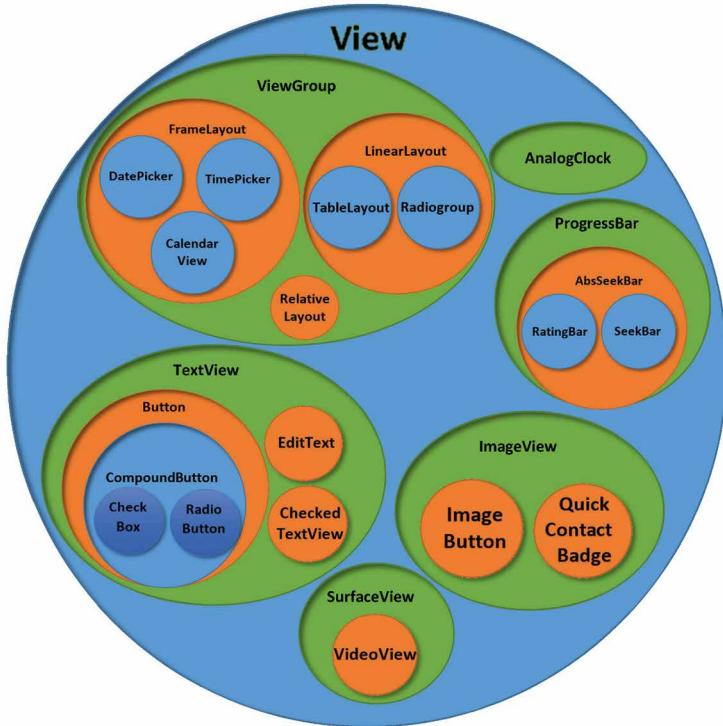


Figura 116 – Hierarquia da classe view

Adaptada de: <https://shre.ink/M6ET>. Acesso em: 6 mar. 2025.



Saiba mais

Para uma referência completa da classe view, consulte o site:

DEVELOPERS. View. [s.d.].j. Disponível em: <https://shre.ink/M6EU>. Acesso em: 6 mar. 2025.

Para entendermos a construção utilizando view, montaremos a tela da figura 117.



Figura 117 – Exemplo de tela de um view simples

O design da montagem do exemplo é feito conforme a figura 118.

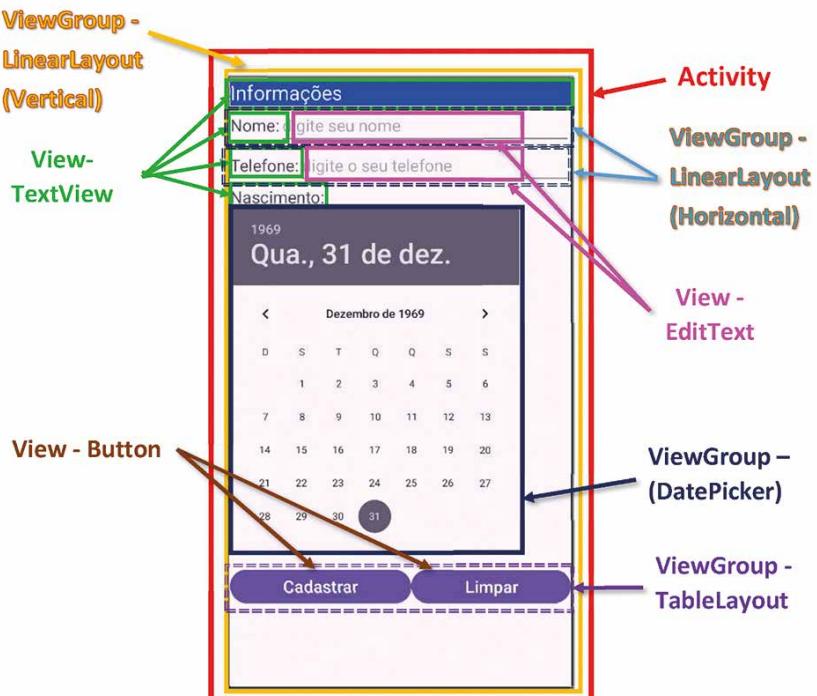


Figura 118 – Design do app da figura 117

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Os elementos foram construídos de acordo com a hierarquia mostrada na figura 116. A tela de fundo é preenchida com um LinearLayout vertical. O importante agora é saber que os elementos colocados abaixo dele serão posicionados um abaixo do outro.

Os elementos abaixo do LinearLayout incluem o texto informações. Abaixo dele, há duas estruturas LinearLayout Horizontal que organizam seus elementos internos lado a lado, o texto Nascimento, um DatePicker para selecionar a data e uma estrutura TableLayout, que distribui os elementos em formato de grade.

Hierarquicamente, o LinearLayout (vertical) contém todos os elementos. Abaixo dele, no mesmo nível, estão o texto informações, os dois LinearLayouts Horizontais que receberão as entradas de informações, o texto Nascimento, o DatePicker e o TableLayout. No terceiro nível, estão os componentes internos dos layouts. Abaixo do primeiro LinearLayout, tem-se o texto Nome e o EditText para a leitura do nome, dispostos lado a lado. Abaixo do segundo LinearLayout, temos o texto Telefone e o EditText para receber o número do telefone. Abaixo do TableLayout, estão os botões Ok e Limpar, cada um em uma célula.

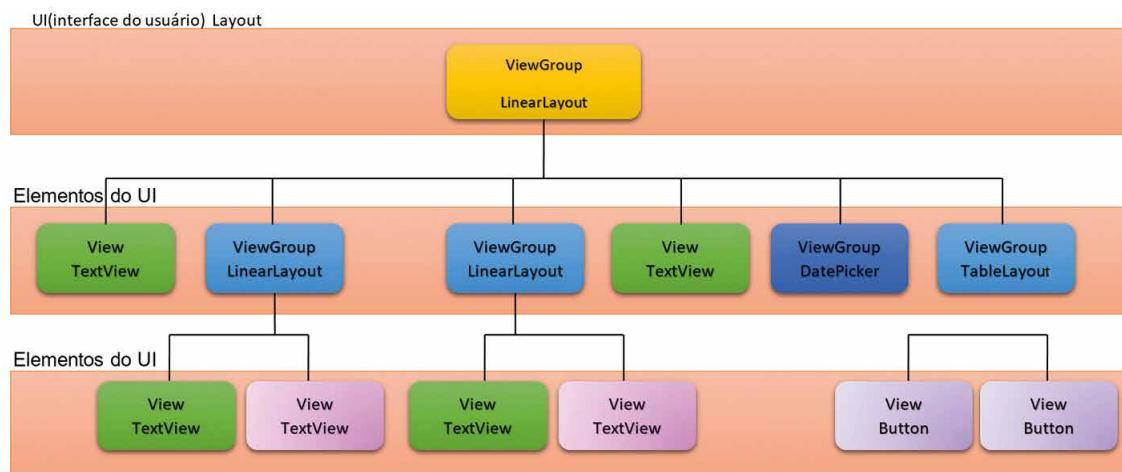


Figura 119 – Hierarquia dos elementos que compõem o design da figura 118

O Android permite criar layouts de forma rápida e intuitiva utilizando XML. Assim como em HTML, define-se a estrutura da sua interface alinhando elementos dentro de outros. Cada layout XML deve ter um elemento raiz, que pode ser um ViewGroup (como LinearLayout ou RelativeLayout) ou uma view (como TextView ou Button). A partir desse elemento raiz, adicionam-se outros elementos, criando uma hierarquia que define a organização e o posicionamento dos elementos na tela. Por exemplo, cria-se um layout vertical com um TextView acima de um Button.

Para montar o layout da figura 118 utilizando apenas o editor de código, a tag raiz do layout, geralmente um ViewGroup como LinearLayout, define a disposição geral dos elementos na tela. Ao configurar o atributo orientation como vertical nessa tag raiz, indica-se que os elementos-filhos desse layout serão organizados verticalmente, um abaixo do outro, seguindo a ordem em que são declarados no código XML.

Unidade II

```
<LinearLayout  
... android:orientation="vertical"  
>  
. .  
</LinearLayout>
```

Seguindo a disposição dos elementos, a primeira linha é um texto definido pelo componente TextView em um layout Android. O TextView exibirá o texto Informações com fonte branca sobre um fundo azul. Ele ocupará toda a largura disponível do layout-pai (match_parent) e ajustará sua altura para acomodar o texto (wrap_content). O tamanho do texto é 24sp, uma unidade que se ajusta à densidade da tela, garantindo legibilidade em diferentes dispositivos. Essa tag é encerrada com />, essa sintaxe é amplamente utilizada para definir elementos da interface do usuário que não contêm texto ou outros elementos-filhos.

```
<TextView  
    android:id="@+id/textView2"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:background="#022EFF"  
    android:text="Informações"  
    android:textColor="#FAFAFA"  
    android:textSize="24sp" />
```

Nas próximas duas linhas, temos o <LinearLayout> definindo um contêiner de layout que organiza seus elementos-filhos em uma única linha horizontal. O atributo ocupa toda a largura disponível do elemento-pai, ajustando a altura do LinearLayout ao conteúdo que ele contém. Esse tipo de layout é útil para organizar elementos de forma linear e horizontal na interface do usuário.

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

Dentro do primeiro LinearLayout estarão enfileirados dois elementos de interface do usuário em um layout: um TextView e um EditText. O TextView exibirá o texto "Nome:" com um tamanho de fonte de 20sp. O EditText permitirá ao usuário inserir seu nome. Ele ocupará toda a largura disponível do layout-pai (match_parent) e ajustará sua altura para acomodar o texto inserido (wrap_content). O atributo android:id="@+id/etNome" atribui um ID ao EditText, permitindo que seja referenciado e manipulado no código Java/Kotlin. O atributo android:hint="digite seu nome" exibe um texto de dica dentro do EditText, indicando ao usuário o tipo de informação a ser inserida.

```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Nome:"  
    android:textSize="20sp" />
```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
<EditText  
    android:id="@+id/etNome"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:textSize="20sp"  
    android:hint="digite seu nome"/>
```

No segundo, o TextView exibirá o texto Telefone sendo identificado como etTel.

```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Telefone:"  
    android:textSize="20sp" />  
  
<EditText  
    android:id="@+id/etTel"  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:textSize="20sp"  
    android:hint="digite o seu telefone"/>
```

Abaixo das duas ListView horizontais consta mais um texto Nascimento.

```
<TextView  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Nascimento:"  
    android:textSize="20sp" />
```

Abaixo, um DatePicker, que é um componente de interface do usuário utilizado para selecionar uma data. O DatePicker ocupará apenas o espaço necessário para exibir a data selecionada (wrap_content para ambas as dimensões). O atributo android:id="@+id/dpNascimento" confere um ID ao DatePicker.

```
<DatePicker  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:id="@+id/dpNascimento"/>
```

O último elemento do LinearLayout vertical é um layout de tabela (TableLayout) com uma única linha (TableRow). O TableLayout organiza seus elementos-filhos em linhas e colunas, semelhantes a uma tabela. Nesse caso, a linha contém dois botões: Cadastrar e Limpar. Ambos os botões ocupam a mesma largura da tela, graças ao atributo android:layout_weight="1". Esse atributo distribui o espaço disponível entre os botões de forma igualitária. O atributo android:layout_gravity="center_horizontal" garante que os botões sejam centralizados horizontalmente dentro da linha.

Unidade II

```

<TableLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
    <TableRow>
        android:layout_height="wrap_content"
        android:layout_width="match_parent"
        android:layout_gravity="center_horizontal">
            <Button
                android:layout_weight="1"
                android:text="Cadastrar"
                android:id="@+id/btOk"
                android:textSize="20sp" />
            <Button
                android:layout_weight="1"
                android:text="Limpar"
                android:id="@+id/btLimpar"
                android:textSize="20sp" />
        </TableRow>
    </TableLayout>

```

Desta forma, o código se completa com os elementos da tela conforme a figura 120.

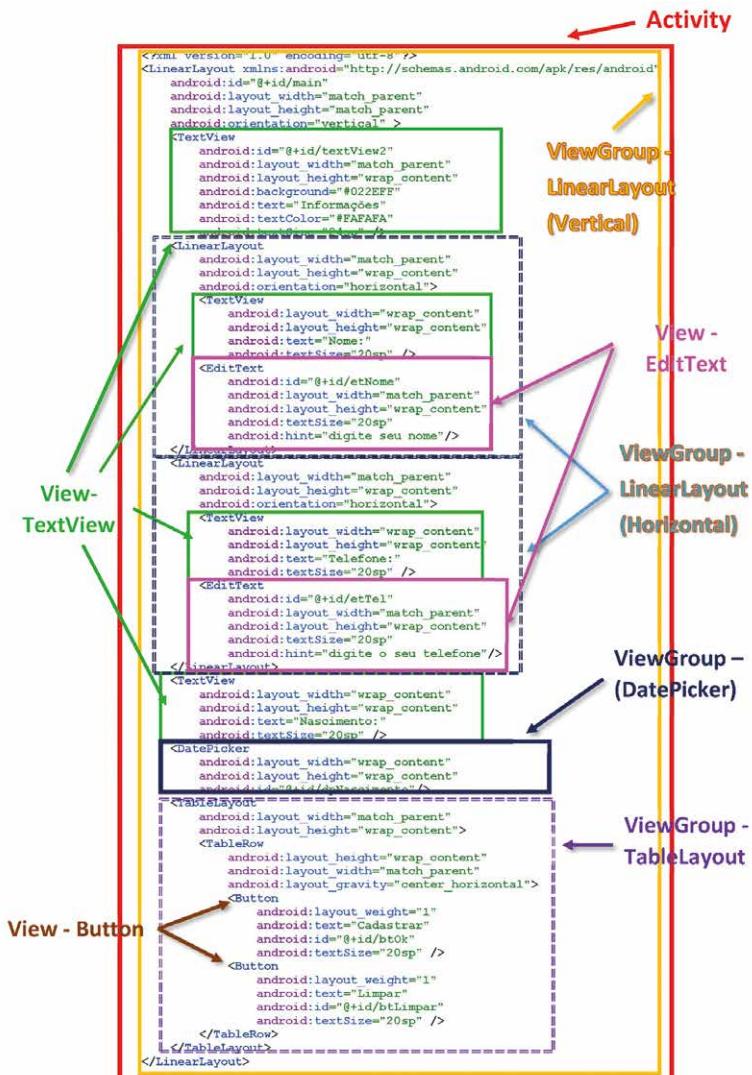


Figura 120 – Distribuição dos elementos no layout

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

A endentação ocorre quando há um recuo em determinadas linhas de código para destacar seu alinhamento em relação a outras linhas. Ela é essencial para a organização visual do código, melhorando legibilidade e hierarquia, o que facilita a compreensão da estrutura dos blocos.

O diagrama ilustra a estrutura hierárquica de um código XML de interface de usuário. A estrutura é representada por uma árvore com ramos coloridos que se ramificam conforme o nível de profundidade da hierarquia. O código XML é o seguinte:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <TextView
        android:id="@+id/textView2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#022EFF"
        android:text="Informações"
        android:textColor="#F0F0F0"
        android:textSize="24sp" />
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Nome:"
            android:textSize="20sp" />
        <EditText
            android:id="@+id/etNome"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textSize="20sp"
            android:hint="digite seu nome" />
    </LinearLayout>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Nome:"
            android:textSize="20sp" />
        <EditText
            android:id="@+id/etNome"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textSize="20sp"
            android:hint="digite seu nome" />
    </LinearLayout>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Telefone:"
            android:textSize="20sp" />
        <EditText
            android:id="@+id/etTel"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:textSize="20sp"
            android:hint="digite o seu telefone" />
    </LinearLayout>
    <Textview
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Nascimento:"
        android:textSize="20sp" />
    <DatePicker
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/dpNascimento" />
    <TableLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >
        <TableRow
            android:layout_height="wrap_content"
            android:layout_width="match_parent"
            android:layout_gravity="center_horizontal" >
            <Button
                android:layout_weight="1"
                android:text="Cadastrar"
                android:id="@+id/btOk"
                android:textSize="20sp" />
            <Button
                android:layout_weight="1"
                android:text="Limpar"
                android:id="@+id/btLimpar"
                android:textSize="20sp" />
        </TableRow>
    </TableLayout>
</LinearLayout>
```

Figura 121 – Distribuição hierárquica do código



Saiba mais

Para uma introdução completa, veja como desenvolver uma interface com visualizações em:

DEVELOPERS. *Desenvolver uma interface com visualizações.* [s.d.]d.
Disponível em: <https://shre.ink/M6YG>. Acesso em: 6 mar. 2025.

A seguir, veremos os componentes de layout mais comuns usados em aplicativos.

6.3.1 ListView

O ListView é um componente essencial no desenvolvimento de aplicativos Android, usado para exibir listas de itens de maneira organizada. Ele atua como um contêiner para apresentar uma quantidade variável de dados em formato de lista, como contatos, itens de menu ou resultados de pesquisa. Altamente personalizável, o ListView permite definir o layout de cada item, adicionar eventos de clique e rolagem, e integrá-lo com outras funcionalidades do aplicativo. Os itens são automaticamente inseridos usando um Adapter, que obtém o conteúdo de uma fonte, como um array ou banco de dados, criando uma visualização para cada entrada de dados. O ListView se encontra no grupo Legacy da paleta.

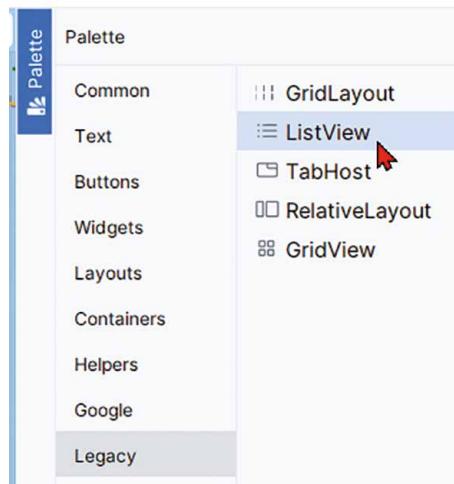


Figura 122

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Para incluir utilizando diretamente o código xml, basta entrar com:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

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

</LinearLayout>
```

Figura 123

Ao alterar o xml, a tela de design exibirá a tela do ListView.

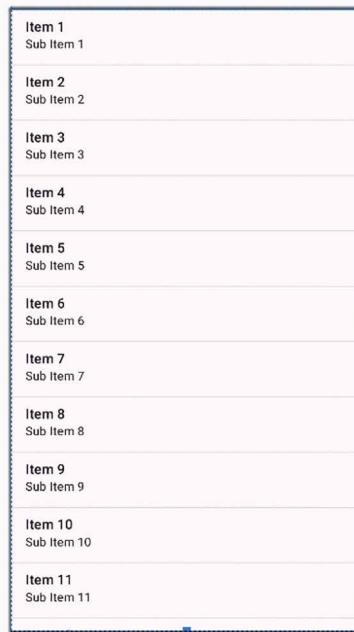


Figura 124 – Tela do ListView

Para instanciar o ListView para ser utilizado no MainActivity, fazemos dentro do ponto do ciclo de vida onCreate():

```
val listView = findViewById<ListView>(R.id.listView)
```

O comando encontra o ListView no layout do aplicativo usando o ID ListView e o atribui à variável ListView.



O layout no exemplo é LinearLayout, e não Constraint Layout. Essa alteração é feita facilmente utilizando o Component Tree abaixo da paleta.

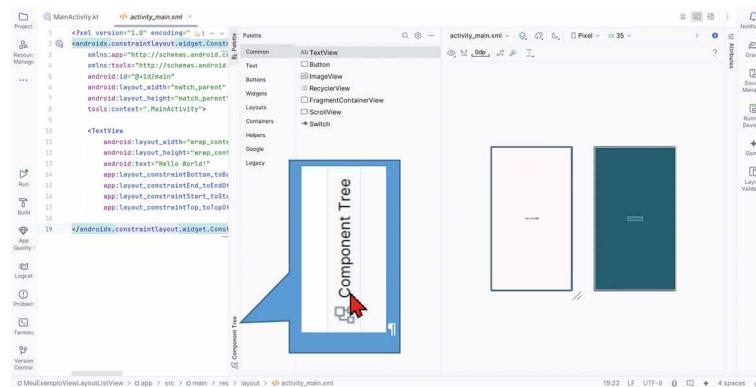


Figura 125

Na raiz main, clicamos com o botão direito do mouse e escolhemos Convert view.

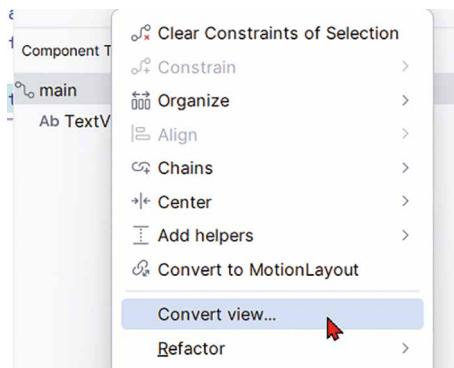


Figura 126

Trocamos para LinearLayout.



Figura 127

Eventualmente com caso de erro basta colocar a orientação para vertical.

Um adapter é um intermediário entre o componente da interface do usuário e a fonte de dados. Ele armazena os dados e os envia para a view do adapter, que obtém os dados do adapter e os exibe em diferentes tipos de views, como ListView, GridView e Spinner. Desta forma preencheremos os dados em um ListView, usando adapter. Itens de lista são inseridos automaticamente em uma lista usando um adapter que puxa o conteúdo de uma fonte. No caso aqui é um array.

Cria-se um array de strings chamado dados, que contém os nomes de vários países e alguns itens adicionais.

```
val dados = arrayOf("Argentina",
    "Bolívia",
    "Brasil",
    "Chile",
    "Colômbia",
    "Equador",
    "Peru",
    "Uruguai",
    "Venezuela",
    "Item 9",
    "Item 10")
```

A seguir criamos um ArrayAdapter chamado adapter, que é um tipo de adaptador que converte o array de strings em visualizações que podem ser exibidas no ListView.

```
val adapter = ArrayAdapter(this, android.R.layout.simple_list_item_1, dados)
```

Como parâmetros do array adapter, temos:

- **this**: refere-se ao contexto atual (activity).
- **android.R.layout.simple_list_item_1**: define o layout padrão do Android para cada item da lista (TextView com o texto na primeira linha).
- **dados**: array de strings contendo os nomes dos países.

Finalmente, define-se o adaptador do ListView como o ArrayAdapter criado. Isso faz com que o ListView exiba os itens dados do array.

```
listView.adapter = adapter
```

O código completo da MainActivity.kt é:

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContentView(R.layout.activity_main)
        val listView = findViewById<ListView>(R.id.listView)
```

```
    val dados = arrayOf("Argentina",
                        "Bolivia",
                        "Brasil",
                        "Chile",
                        "Colombia",
                        "Equador",
                        "Peru",
                        "Uruguai",
                        "Venezuela",
                        "Item 9",
                        "Item 10")
    val adapter = ArrayAdapter(this,
                                android.R.layout.simple_list_item_1, dados)
    listView.adapter = adapter
}
}
```



Saiba mais

Consulte a referência para a classe ListView:

DEVELOPERS. *ListView*. [s.d.]g. Disponível em: <https://shre.ink/M6PM>. Acesso em: 6 mar. 2025.

6.3.2 TextView (visualização de texto)

O TextView é um componente fundamental na interface do usuário de aplicativos Android, utilizado para exibir texto na tela. Ele personaliza a aparência do texto, como fonte, tamanho, cor e alinhamento. Ele é configurado para responder a eventos como cliques. Além de exibir texto simples, o TextView suporta funcionalidades mais avançadas, como formatação de texto, links clicáveis e ajuste automático do tamanho da fonte.

No arquivo xml, ele é definido pela tag:

```
<TextView ... />
```

No momento em que um novo Empty View Project é criado, um TextView mostrando Hello World já vem pré-criado.

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Hello World!"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Executando o emulador, temos:

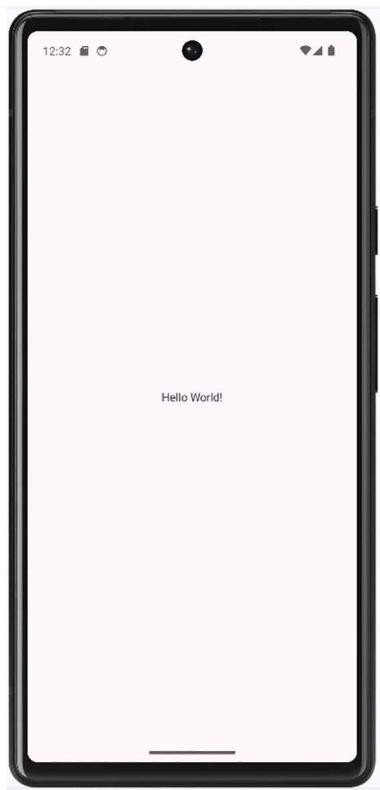


Figura 128

É possível alterar o texto no próprio código do Kotlin. Para manipular o elemento, acrescentamos uma identificação na tag do TextView existente:

```
android:id="@+id/txtTexto"
```

Chamaremos esse elemento de texto de txtTexto. Ele será usado para conectar o design (XML) com a lógica (Kotlin). No arquivo MainActivity.kt exclui-se a chamada da função ViewCompat, pois neste momento ele é desnecessário. Acrescentaremos as seguintes linhas de comando:

```
val txtTexto = findViewById<TextView>(R.id.txtTexto)
```

É declarada uma variável imutável chamada txtTexto para armazenar a referência ao elemento TextView, que é encontrado no layout pelo ID txtTexto.

```
txtTexto.setTextSize(30f)
```

O método setTextSize define o tamanho do texto em sp (unidade de pixels escalada). No caso do TextView referenciado por txtTexto, o tamanho do texto é alterado para 30 sp.

```
txtTexto.text = "Texto alterado"
```

Unidade II

O método `text` modifica o conteúdo textual exibido pelo `TextView` chamado `txtTexto`. No caso, a string `Texto alterada` é definida como o novo texto a ser exibido.

O resultado a ser alterado é:



Figura 129

O código xml alterado é:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        android:id="@+id/txtTexto"
    />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Figura 130

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

O código Kotlin alterado é:

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        enableEdgeToEdge()  
        setContentView(R.layout.activity_main)  
        val txtTexto = findViewById<TextView>(R.id.txtTexto)  
        txtTexto.setTextSize(30f)  
        txtTexto.text = "Texto alterado"  
    }  
}
```

Figura 131

Veja um exemplo completo:

```
<TextView  
  
    android:layout_width="wrap_content" android:layout_height="wrap_content"  
    android:text="Text exemplo"  
  
/>
```



Figura 132

6.3.3 Menu

Os recursos de menu são uma parte da interface do usuário de aplicativos Android. Eles permitem a criação de menus personalizados e organizados de forma eficiente. O MenuInflater é a ferramenta para transformar esses recursos XML em objetos do menu que podem ser exibidos na tela. Existem três tipos principais de menus:

- **Menu de opções:** menu principal de uma atividade, no qual são inseridas ações com impacto global no aplicativo, como configurações ou sobre. Ele geralmente aparece na barra de ação do aplicativo.
- **Menu de contexto:** menu flutuante que aparece quando o usuário realiza um toque longo em um elemento. Ele oferece ações que afetam o conteúdo selecionado.

- **Menu pop-up:** exibe uma lista vertical de itens ancorados à visualização que invoca o menu. É útil para fornecer ações relacionadas a um conteúdo específico.

Para entendermos o processo, criaremos um menu de opções.



Figura 133 – Menu simples com duas opções

O Android oferece um mecanismo eficiente e padronizado para criar menus por meio de arquivos XML. Em vez de definir a estrutura de um menu diretamente no código da sua atividade, centraliza-se essa informação em um arquivo XML específico. Isso separa a lógica da sua aplicação da sua interface visual, tornando o código organizado e facilitando a manutenção. Por padrão, o arquivo com um menu está localizado na pasta menu dentro da pasta res. Caso essa pasta não exista, deve-se criá-la.

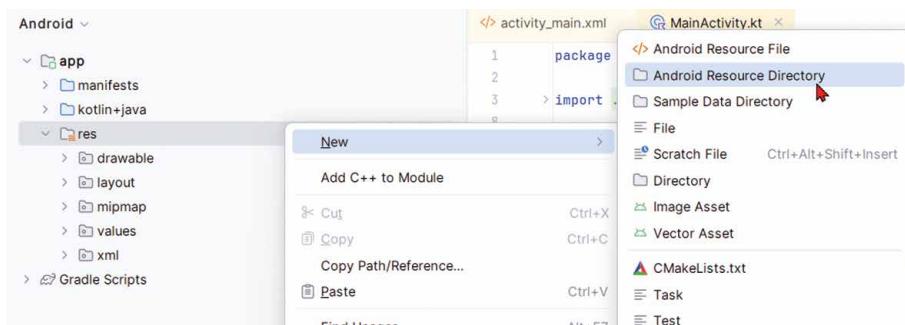


Figura 134

Na pasta res, clica-se com o botão da direita e, em new, escolhe-se Android Resource Directory. Abrirá uma tela para definir o tipo da pasta, clicar em resource type e escolher menu.

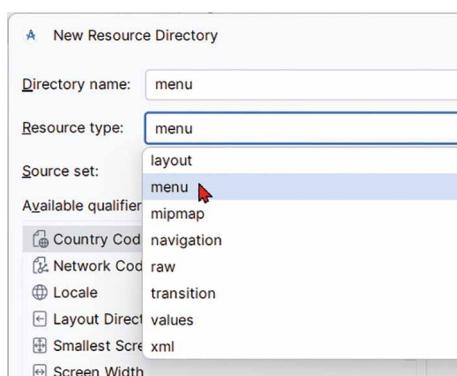


Figura 135

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Uma vez criada a pasta, crie-se a pasta xml. Agora clique com o botão direito na pasta menu e, em new, escolha menu resource file.



Figura 136

Ao abrir a tela do New Resource File, identifique o novo arquivo com o nome da pasta, no caso meu_menu.

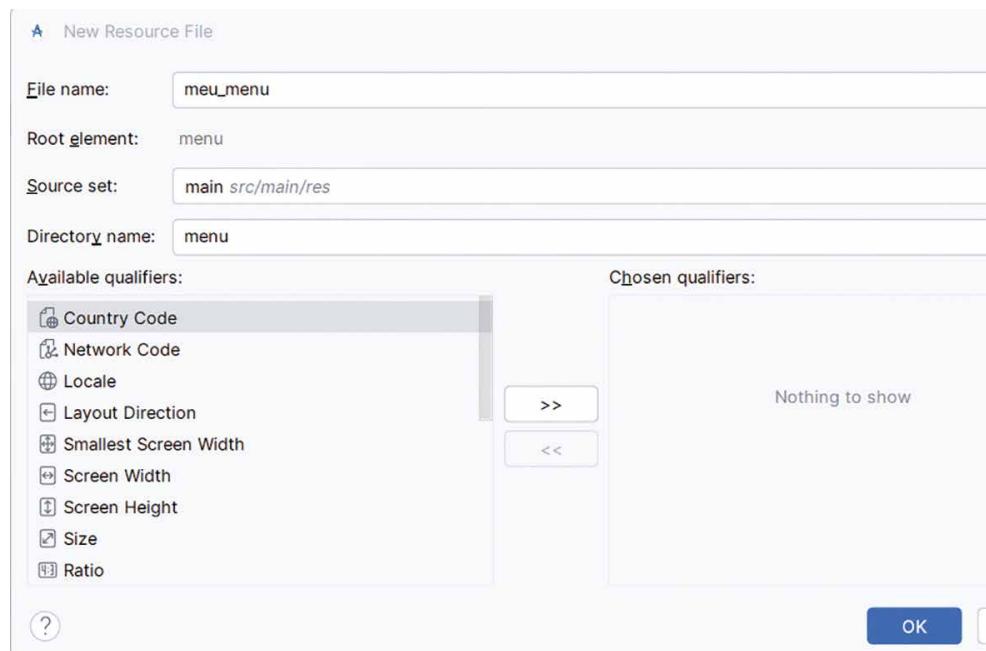


Figura 137

Unidade II

O arquivo está pronto para ser editado. Na tela do editor, abre-se a janela do projeto e o design. Caso não apareça, clique sobre o arquivo na árvore de projetos e escolha a visualização em Split.

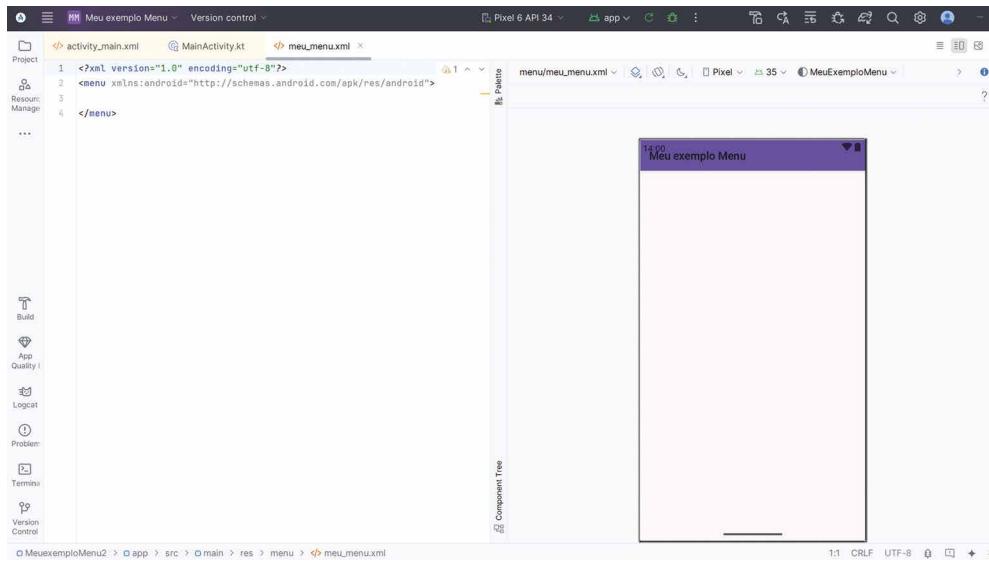


Figura 138

Observe na tela do editor que a tag menu está criada. Dentro da tag, cria-se o primeiro item. Vamos apresentar o texto Configuração e identificá-lo como ItemConfiguracao.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/ItemConfiguracao"
        android:title="Configuração"
        app:showAsAction="never"
    />
</menu>
```

Figura 139

Observe que no designer surgiu um item.



Figura 140

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Criaremos mais um item, que apresentará o texto Sobre e identificamos como ItemSobre.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto">
    <item
        android:id="@+id/ItemConfiguracao"
        android:title="Configuração"
        app:showAsAction="never"
    />
    <item
        android:id="@+id/ItemSobre"
        android:title="Sobre"
        app:showAsAction="never"
    />

</menu>
```

Figura 141

A tela do design é atualizada para:



Figura 142

Para apresentar uma barra de ferramentas no alto do nosso aplicativo, vamos inserir um toolbar ou barra de ferramentas. Isso oferece acesso rápido e conveniente a funcionalidades e recursos específicos de um software ou aplicativo. Ele inclui botões para navegação, pesquisa, favoritos, configurações, notificações e integração com outros aplicativos. No caso, conterá a chamada para o menu. No exemplo utilizaremos um layout que está na biblioteca Material Design chamado AppBarLayout, interno a ele, um widget Toolbar.

```
<com.google.android.material.appbar.AppBarLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    tools:ignore="MissingConstraints">
    <androidx.appcompat.widget.Toolbar
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/toolbar"/>
</com.google.android.material.appbar.AppBarLayout>
    android:id="@+id/toolbar"/>
```

A tela do designer fica:

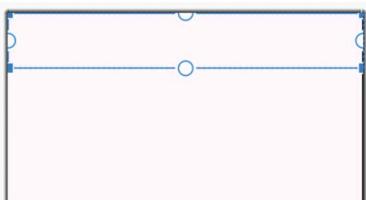


Figura 143

Agora conectaremos o código com as telas.

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val toolbar: androidx.appcompat.widget.Toolbar =
            findViewById(R.id.toolbar)
        setSupportActionBar(toolbar)
    }
}
```

Figura 144

Na linha:

```
val toolbar: androidx.appcompat.widget.Toolbar = findViewById(R.id.toolbar)
```

A variável chamada toolbar inicia-se com a localização feita pelo findViewById(), buscando pela identidade toolbar (R.id.toolbar) no activity_main.xml.

```
setSupportActionBar(toolbar)
```

Esse método define a toolbar como a barra de ação da atividade. Isso significa que a toolbar permite todas as funcionalidades como menus e navegação.

Para configurar o menu de opções de uma atividade no Android, utiliza-se o método onCreateOptionsMenu. Dentro desse método, o MenuInflater infla o layout do menu a partir de um arquivo XML (meu_menu.xml), localizado na pasta res/menu do projeto. O menu inflado é associado ao objeto menu passado como parâmetro. Retornar true indica que o menu foi criado com sucesso e deve ser exibido na interface do usuário. Esse método é essencial para definir e exibir as opções de menu personalizadas em uma atividade.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

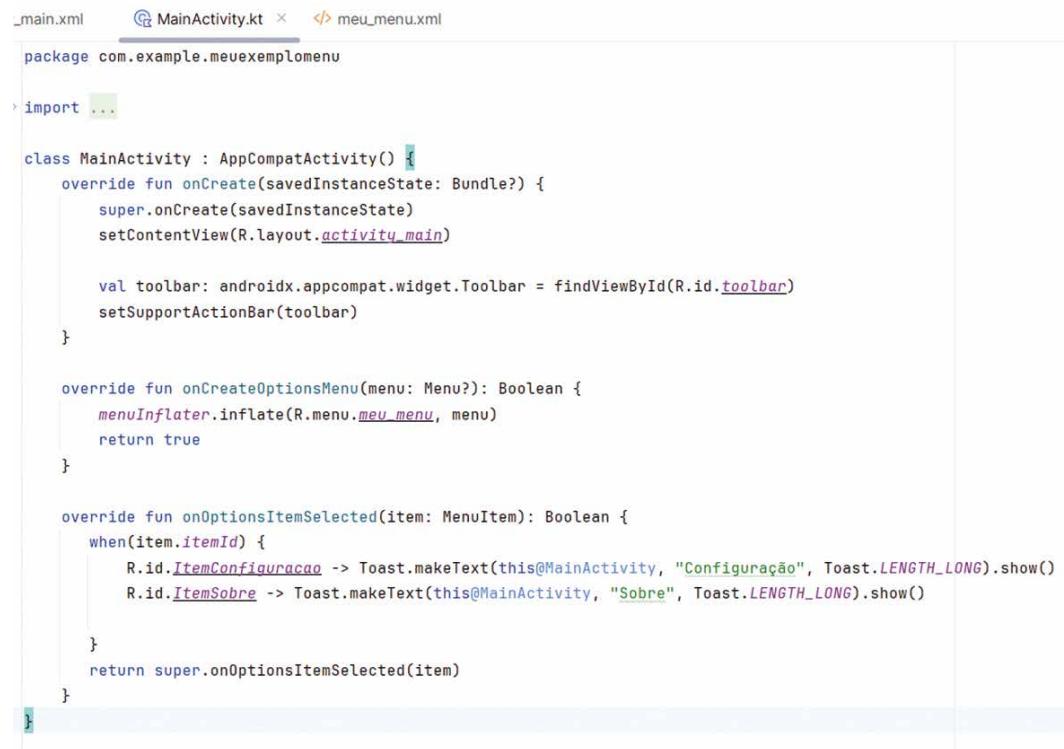
```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {  
    menuInflater.inflate(R.menu.meu_menu, menu)  
    return true  
}
```

Para tratar a seleção de itens no menu de opções de uma atividade, utilizamos o método `onOptionsItemSelected`.

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {  
    when(item.itemId) {  
        R.id.ItemConfiguracao ->  
            Toast.makeText(this@MainActivity, "Configuração",  
                Toast.LENGTH_LONG).show()  
        R.id.ItemSobre -> Toast.makeText(this@MainActivity, "Sobre",  
            Toast.LENGTH_LONG).show()  
  
    }  
    return super.onOptionsItemSelected(item)  
}
```

Utilizando uma estrutura `when`, ele verifica o ID do item selecionado (`item.itemId`) e executa ações específicas para cada caso. Se o item selecionado for `R.id.ItemConfiguracao`, ele exibe uma mensagem configuração usando um toast. Da mesma forma, se o item selecionado for `R.id.ItemSobre`, ele exibe uma mensagem sobre. Após tratar os itens, o método retorna à chamada para a implementação da superclasse (`super.onOptionsItemSelected(item)`), garantindo que qualquer item não processado seja tratado.

O código do `MainActivity` fica:



```
_main.xml  ↗ MainActivity.kt  ✘ meu_menu.xml  
package com.example.meuexemplomenu  
  
> import ...  
  
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        val toolbar: androidx.appcompat.widget.Toolbar = findViewById(R.id.toolbar)  
        setSupportActionBar(toolbar)  
    }  
  
    override fun onCreateOptionsMenu(menu: Menu?): Boolean {  
        menuInflater.inflate(R.menu.meu_menu, menu)  
        return true  
    }  
  
    override fun onOptionsItemSelected(item: MenuItem): Boolean {  
        when(item.itemId) {  
            R.id.ItemConfiguracao -> Toast.makeText(this@MainActivity, "Configuração", Toast.LENGTH_LONG).show()  
            R.id.ItemSobre -> Toast.makeText(this@MainActivity, "Sobre", Toast.LENGTH_LONG).show()  
  
        }  
        return super.onOptionsItemSelected(item)  
    }  
}
```

Figura 145

Ao executar:

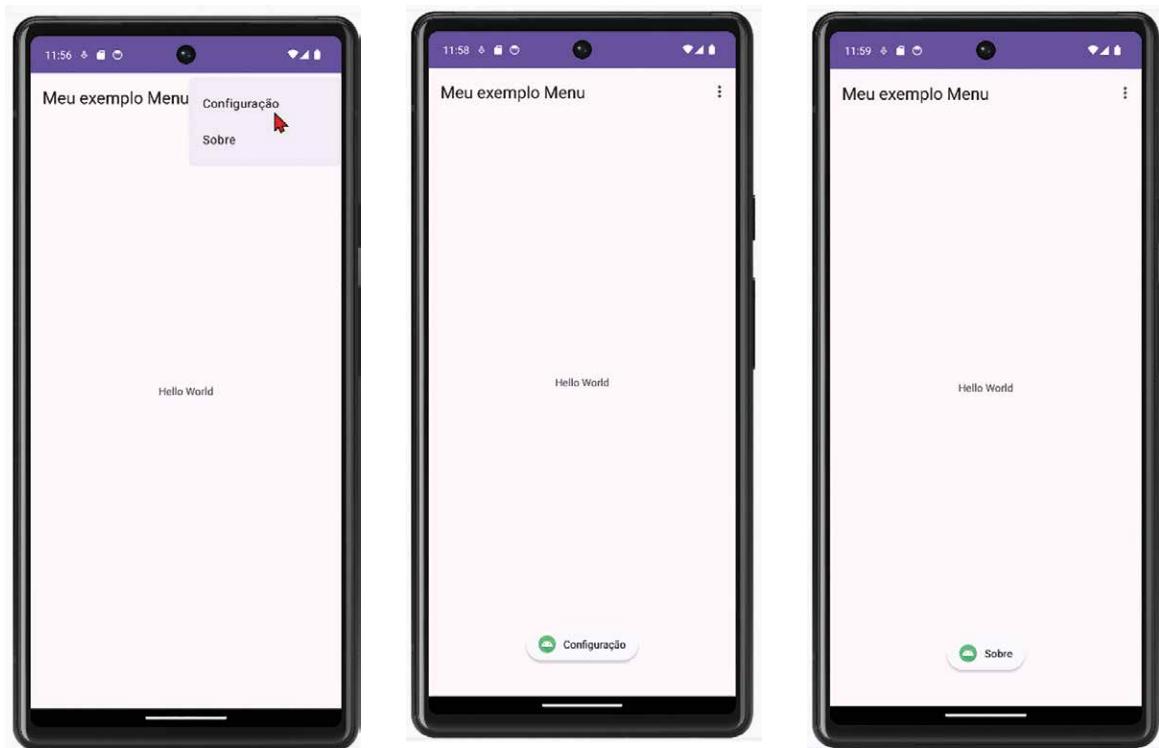


Figura 146

Observação

O toast é uma maneira simples e rápida de exibir mensagens temporárias na tela de um aplicativo Android. Ele aparece como um pequeno pop-up que ocupa apenas o espaço necessário para a mensagem, sem interromper a interação do usuário com a atividade atual. O toast desaparece automaticamente após um curto período, tornando-o ideal para fornecer feedback breve e não intrusivo, como confirmações de ações ou notificações rápidas. Para exibir um toast, utiliza-se o método `Toast.makeText()`, passando o contexto, a mensagem e a duração desejada, seguido pelo método `show()` para exibir a mensagem.



Saiba mais

Para mais informações sobre o toast, pesquise em:

DEVELOPERS. Visão geral dos avisos. [s.d.]. Disponível em: <https://shre.ink/MOUa>. Acesso em: 6 mar. 2025.

6.3.4 AlertDialog

O AlertDialog é uma ferramenta que cria caixas de diálogo personalizadas para interagir com o usuário. Uma das suas principais vantagens é que, diferentemente de outras interfaces do Android, não é obrigatório criar um layout XML específico para ele. Agiliza o desenvolvimento, pois define o conteúdo e as opções da caixa de diálogo diretamente no código. Em princípio, ele tem um layout padrão quando programado.

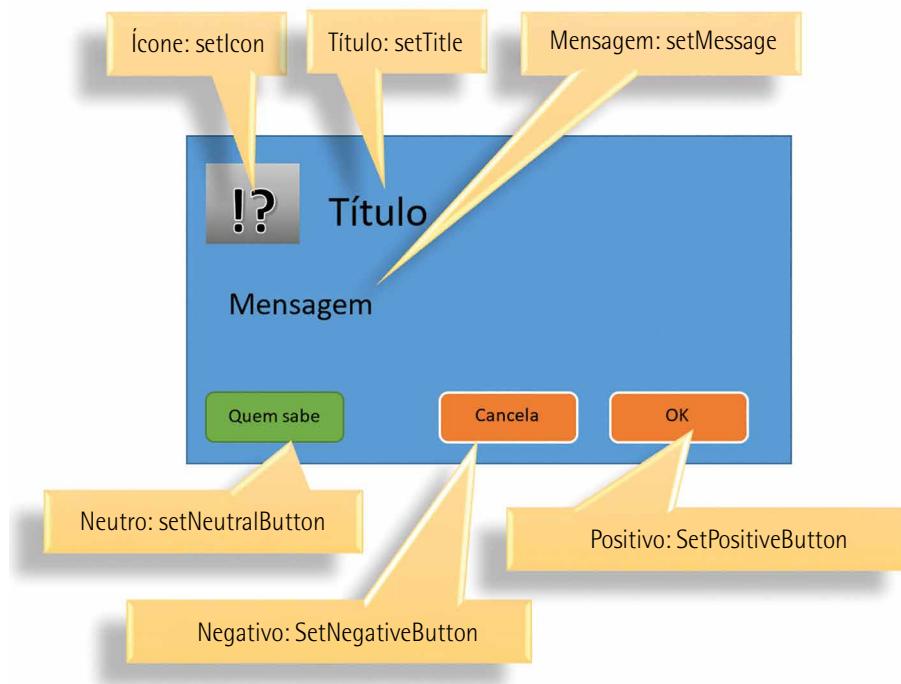


Figura 147 – AlertDialog padrão

O AlertDialog.Builder oferece métodos intuitivos para configurar o título, a mensagem e os botões. Ele adiciona listas ou customiza o layout de forma mais complexa. Os métodos básicos são:

- **setTitle(String)**: define o título da caixa de diálogo.
- **setMessage(String)**: define a mensagem principal da caixa de diálogo.
- **setIcon(int iconId)**: define um ícone para a caixa de diálogo.
- **setPositiveButton(String, DialogInterface.OnClickListener listener)**: adiciona um botão positivo (geralmente ok) com o texto especificado e um listener para o evento de clique.
- **setNegativeButton(String, DialogInterface.OnClickListener listener)**: adiciona um botão negativo (geralmente cancelar) com o texto especificado e um listener para o evento de clique.

- **setNeutralButton(String, DialogInterface.OnClickListener listener)**: adiciona um botão neutro (geralmente neutro) com o texto especificado e um listener para o evento de clique.
- **setCancelable(boolean)**: define se a caixa de diálogo pode ser fechada clicando fora dela.
- **show()**: exibe a caixa de diálogo na tela.

Exemplo:

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        enableEdgeToEdge()  
        setContentView(R.layout.activity_main)  
        val alerta = AlertDialog.Builder(this)  
        alerta.setTitle("Alerta do Android")  
        alerta.setMessage("Uma mensagem para você")  
  
        alerta.setPositiveButton("Certo") { dialog, which ->  
            Toast.makeText(getApplicationContext,  
                "Certo", Toast.LENGTH_SHORT).show()  
        }  
  
        alerta.setNegativeButton("Cancela") { dialog, which ->  
            Toast.makeText(getApplicationContext,  
                "Cancela", Toast.LENGTH_SHORT).show()  
        }  
  
        alerta.setNeutralButton("Quem Sabe") { dialog, which ->  
            Toast.makeText(getApplicationContext,  
                "Talvez", Toast.LENGTH_SHORT).show()  
        }  
  
        alerta.show()  
    }  
}
```

Figura 148

Disponível em: <https://shre.ink/M0Fg>. Acesso em: 6 mar. 2025.



Figura 149

6.3.5 LinearLayout

O LinearLayout é um dos layouts mais básicos e comumente utilizados no desenvolvimento de aplicativos Android. Ele organiza os elementos da interface do usuário de forma linear, seja na horizontal ou na vertical. Essa simplicidade o torna ideal para criar layouts simples e intuitivos.

O LinearLayout oferece grande flexibilidade por atributos como android:orientation, que define a direção dos elementos (horizontal ou vertical), e android:layout_weight, que distribui o espaço disponível entre os elementos de forma proporcional. Além disso, ele pode ser aninhado com outros layouts para criar estruturas mais complexas.

Como exemplo, a partir de um novo View Activity, vamos mudar no activity_main.xml o layout para LinearLayout e deixar a orientação vertical.

Substituiremos o texto Hello World por dois textos e um botão. O xml fica assim:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Texto 1"
        android:gravity="center"
        android:textSize="18sp" />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Texto 2"
        android:gravity="center"
        android:textSize="18sp" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Botão 1"
        android:gravity="center" />
</LinearLayout>
```

Figura 150

O LinearLayout principal define a orientação vertical. O atributo layout_weight faz com que cada elemento ocupe 1/3 do espaço vertical disponível, já que todos possuem o mesmo peso.

O atributo gravity centraliza o texto dentro de cada elemento. Temos como resultado a tela dividida em três partes iguais verticalmente. Cada parte contém um dos elementos centralizado.

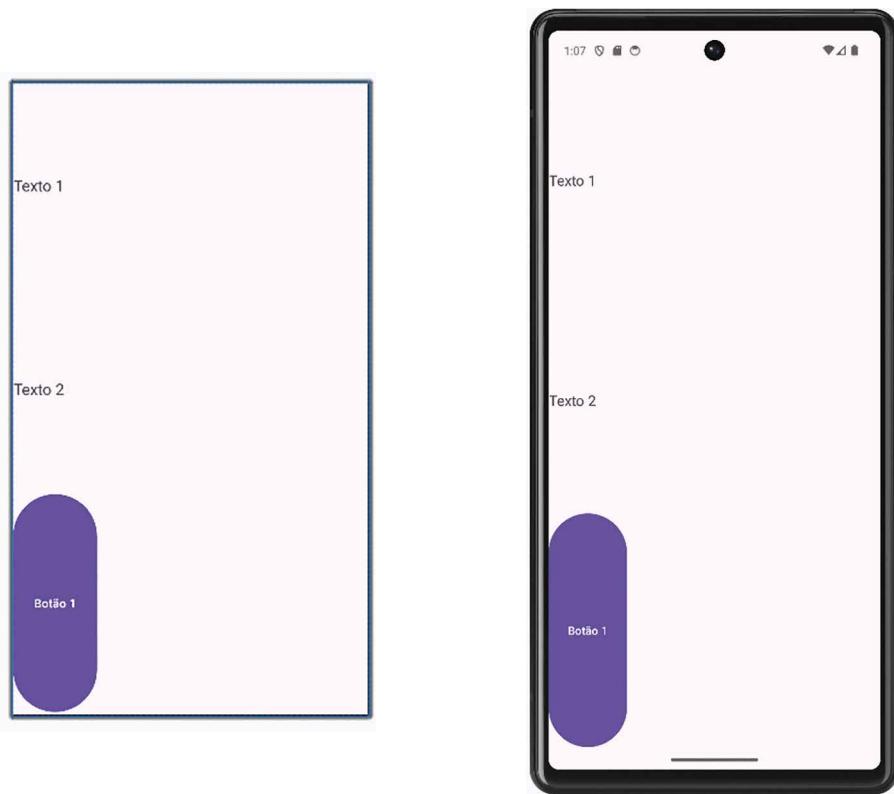


Figura 151

Se alterarmos o atributo da orientação para horizontal:

```
android:orientation="horizontal"
```

Temos o seguinte resultado:

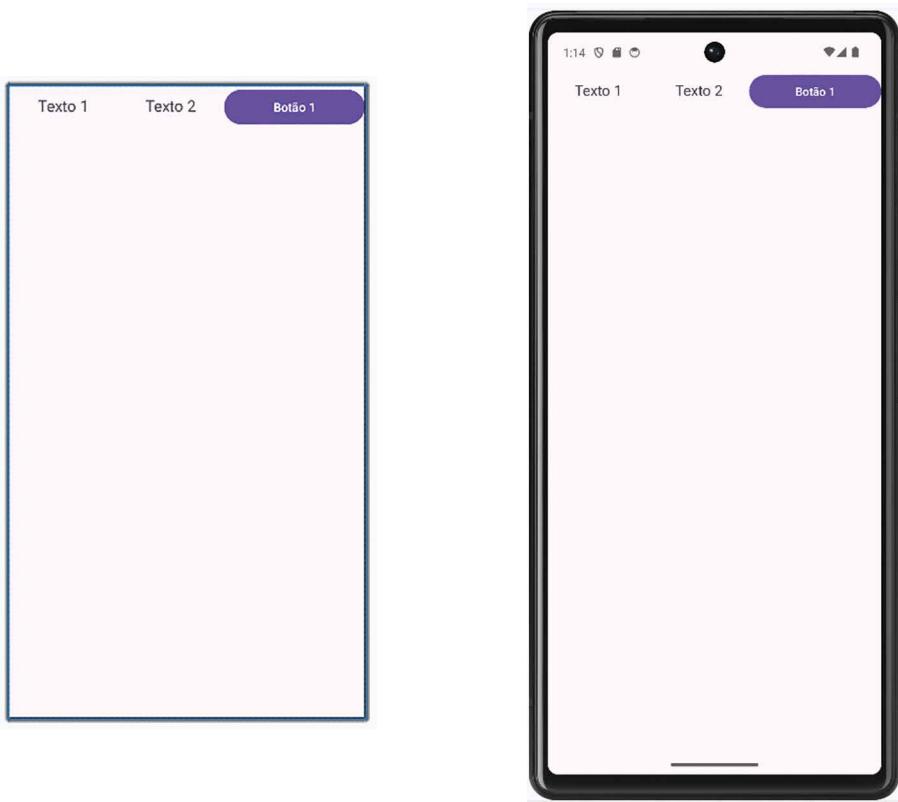


Figura 152

6.3.6 RelativeLayout

O RelativeLayout posiciona elementos da interface do usuário em relação a outros elementos ou aos limites do layout-pai. Isso proporciona flexibilidade na criação de layouts complexos e personalizados. Diferente do LinearLayout, que organiza os elementos em uma ordem linear, o RelativeLayout permite especificar a posição de cada elemento em relação a outros elementos ou aos limites do layout-pai, podendo definir se um elemento estará acima, abaixo, à esquerda, à direita, centralizado ou alinhado a outros elementos por meio dos atributos:

- **android:layout_above**: posiciona o elemento acima de outro elemento.
- **android:layout_below**: posiciona o elemento abaixo de outro elemento.
- **android:layout_toLeftOf**: posiciona o elemento à esquerda de outro elemento.
- **android:layout_toRightOf**: posiciona o elemento à direita de outro elemento.
- **android:layout_alignParentTop**: alinha o elemento ao topo do layout-pai.
- **android:layout_alignParentBottom**: alinha o elemento à base do layout-pai.

- **android:layout_alignParentLeft**: alinha o elemento à esquerda do layout-pai.
- **android:layout_alignParentRight**: alinha o elemento à direita do layout-pai.
- **android:layout_centerVertical**: centraliza o elemento verticalmente dentro do layout-pai.
- **android:layout_centerHorizontal**: centraliza o elemento horizontalmente dentro do layout-pai.

Como exemplo, iniciaremos um novo projeto e alteraremos o layout do activity_main.xml para RelativeLayout. Monta-se a seguinte estrutura com uma disposição pouco convencional para entender o layout:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/text1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:text="Texto 1" />
    <TextView
        android:id="@+id/text2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toLeftOf="@+id/text1"
        android:text="Texto 2" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:text="Botão" />
</RelativeLayout>
```

Figura 153

O primeiro TextView (@+id/text1) tem largura e altura ajustadas ao conteúdo (wrap_content). Ele é alinhado à direita do pai (android:layout_alignParentRight="true") e exibe o texto 1. O segundo TextView (@+id/text2) é posicionado à esquerda do primeiro TextView (android:layout_toLeftOf="@+id/text1") e exibe o texto 2. O botão (button) é centralizado no layout-pai (android:layout_centerInParent="true"). Ele exibe o texto botão. O texto 2 fica localizado antes do texto 1 no layout resultante (figura 154).

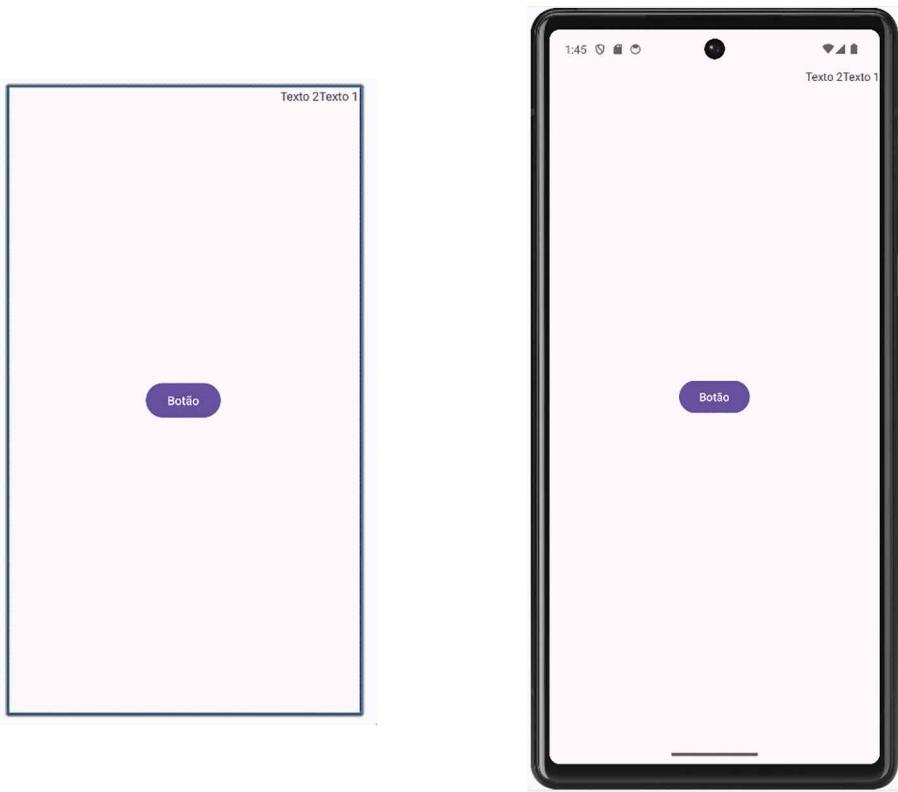


Figura 154

6.3.7 TableLayout

O TableLayout no Android Studio é um contêiner que organiza elementos da interface em uma estrutura tabular, similar às tabelas HTML. Ele divide o conteúdo em linhas (TableRow) e colunas, proporcionando uma forma organizada de apresentar dados. Cada célula da tabela contém diversos tipos de elementos visuais, como textos ou imagens.

Vamos criar um exemplo com o seguinte activity_main.xml:

```
<TableLayout xmlns:android="http://schemas.android.com/apk/res/android"  
    android:id="@+id/main"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
    <TableRow>  
        <TextView  
            android:layout_width="0dp"  
            android:layout_height="wrap_content"  
            android:layout_weight="2"  
            android:text="Coluna 1"  
            android:gravity="center" />  
        <TextView  
            android:layout_width="0dp"  
            android:layout_height="wrap_content"  
            android:layout_weight="2"
```

Unidade II

```
    android:text="Coluna 2"
    android:gravity="center" />
<TextView
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:text="Coluna 3"
    android:gravity="center" />
</TableRow>

<TableRow>
    <TextView
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Linha 1, Coluna 1" />
    <TextView
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Linha 1, Coluna 2" />
    <TextView
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Linha 1, Coluna 3" />
</TableRow>

<TableRow>
    <TextView
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Linha 2 Coluna 1"
        android:gravity="center" />
    <TextView
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="2"
        android:text="Linha 2 Coluna 2"
        android:gravity="center" />
    <TextView
        android:layout_width="0dp"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:text="Linha 2 Coluna 3"
        android:gravity="center" />
</TableRow>

</TableLayout>
```

No exemplo criamos uma tabela com três linhas (TableRow), cada uma com três TextViews:

- Primeira TableRow:
 - O atributo android:layout_weight é usado para distribuir o espaço disponível. Os dois primeiros TextView têm peso 2 e o terceiro tem peso 1, o que significa que os dois primeiros ocuparão mais espaço proporcionalmente.
 - O texto e a gravidade (android:gravity="center") são definidos para centralizar o texto dentro de cada célula.
- Segunda TableRow:
 - Todos os TextView têm peso 1, distribuindo o espaço igualmente entre eles.
 - O texto é definido para cada célula, mas sem gravidade centralizada.
- Terceira TableRow:
 - O primeiro e o terceiro TextView têm peso 1, enquanto o segundo tem peso 2, ocupando mais espaço proporcionalmente.
 - O texto e a gravidade são definidos para centralizar o texto dentro de cada célula.

Como resultado temos:

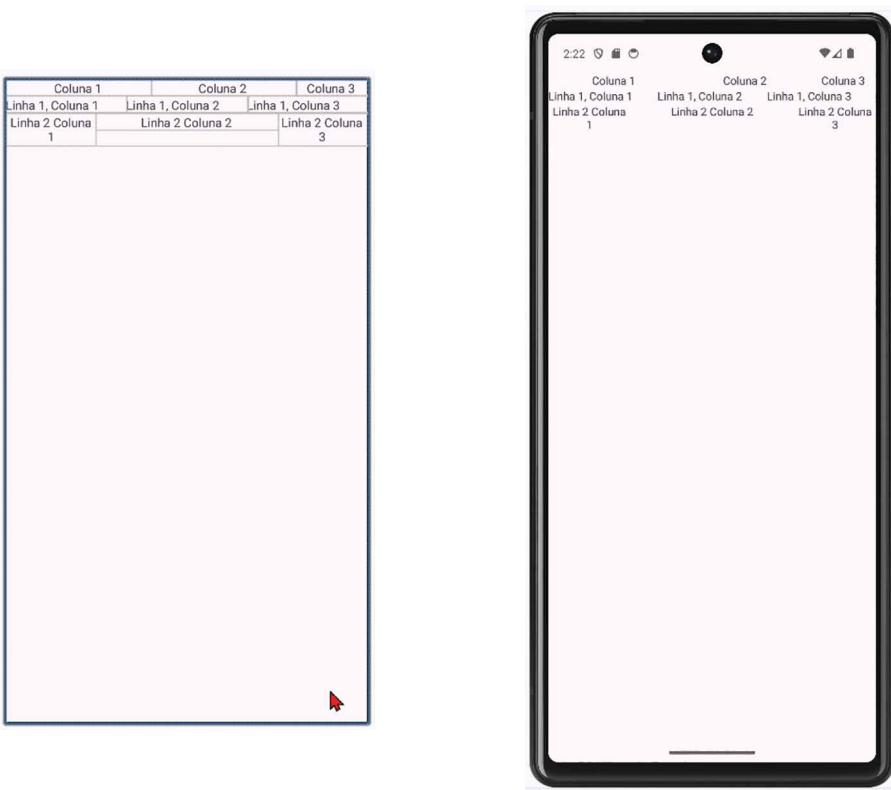


Figura 155

6.4 Jetpack Compose

O Jetpack Compose (ou apenas Compose) é a nova forma de criar interfaces do usuário para Android. Com ele, escreve-se menos código e obtém-se mais resultados, graças à sua abordagem declarativa e componentes pré-construídos. Cria Interfaces de Usuário (UIs) personalizadas e responsivas com facilidade, utilizando o poder do Kotlin.

No Compose, é necessário menos código para desenvolver um aplicativo, o que simplifica todo o ciclo de desenvolvimento. Isso permite que os desenvolvedores se concentrem na lógica central, reduzam o tempo de teste e depuração, e criem códigos mais robustos. Para as equipes, menos código significa menor complexidade, facilitando a manutenção e a colaboração.

A antiga forma de atualizar a interface do Android era manual e complexa. Ao modificar diretamente os elementos da árvore de views, era fácil cometer erros e dificultar a manutenção do código. Essa abordagem, embora funcional, não é ideal para projetos maiores e mais complexos.

Antes de estudar a teoria, vamos criar um projeto inicial de Projeto Vazio Compose (Empty Activity). Todos os exemplos serão desenvolvidos após testarmos e concluirmos a configuração padrão inicial.

Inicie um novo projeto Empty Activity e observe que não se trata de uma atividade de views. Atualmente, por padrão, a atividade vazia (empty activity) utiliza o Compose (figura 156).

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

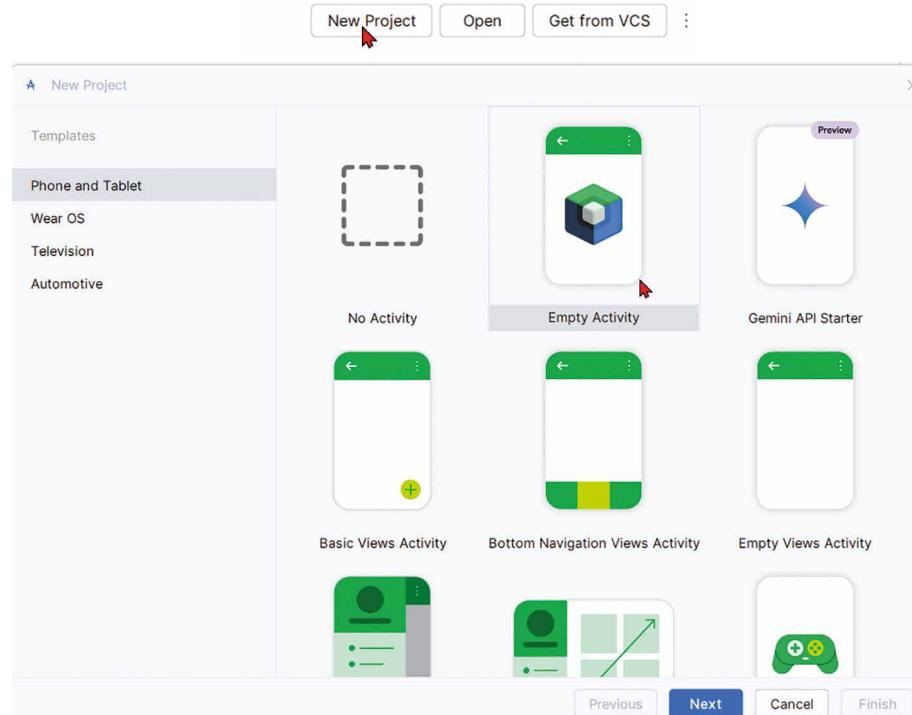


Figura 156

Complete com o nome do projeto. Neste exemplo é "Meu Primeiro Compose", mas nos exemplos posteriores utilizaremos nomes diferentes. Clique em Finish.

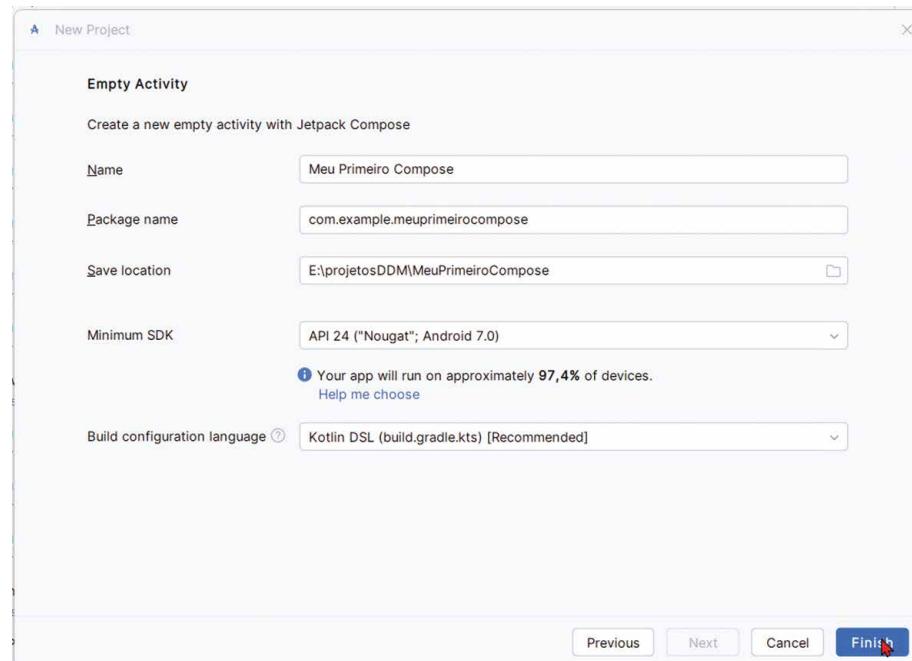
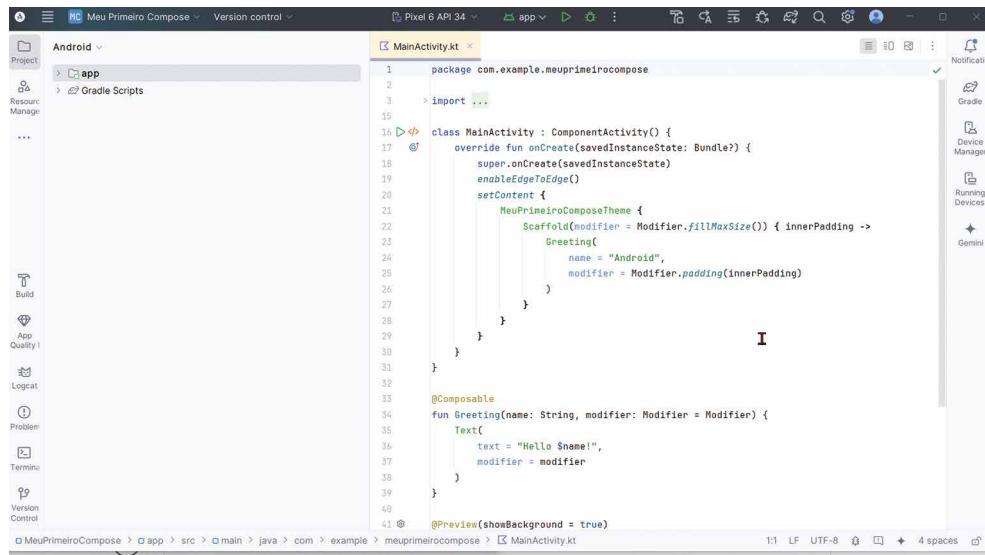


Figura 157

Unidade II

Espere o projeto carregar completamente.



The screenshot shows the Android Studio interface with the project 'Meu Primeiro Compose' open. The code editor displays the `MainActivity.kt` file, which contains Kotlin code for a Compose application. The code defines a `MainActivity` class that overrides `onCreate` and sets the content to a `Greeting` composable. A `@Preview` annotation is used to preview the layout. The right side of the screen shows various toolbars and panels like Notifications, Gradle, Device Manager, and Gemini.

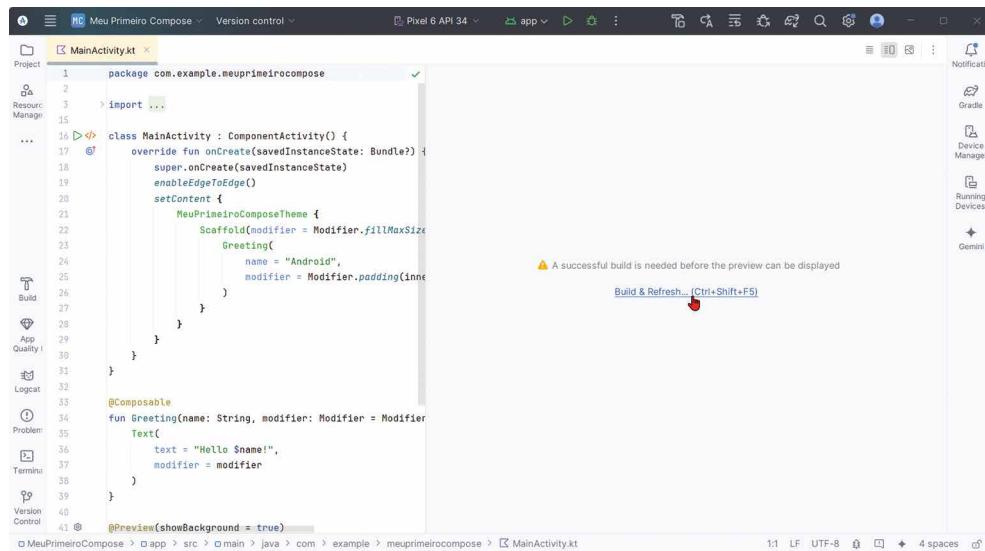
Figura 158

Feche a janela de projeto e escolha a visão split do editor.



Figura 159

No modo Compose, não há arquivo XML e, portanto, não existe um editor de layout. Na tela dividida, é exibida uma amostra do resultado do layout criado com programação declarativa. Para a primeira geração, é necessário processar uma vez. Clique em Build & Refresh.



The screenshot shows the Android Studio interface with the project 'Meu Primeiro Compose' open. The code editor displays the `MainActivity.kt` file. A tooltip message 'A successful build is needed before the preview can be displayed' appears above the `@Preview` annotation. Below the code editor, a red arrow points to the 'Build & Refresh' button in the toolbar. The right side of the screen shows various toolbars and panels like Notifications, Gradle, Device Manager, and Gemini.

Figura 160

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Espere o build terminar completamente:

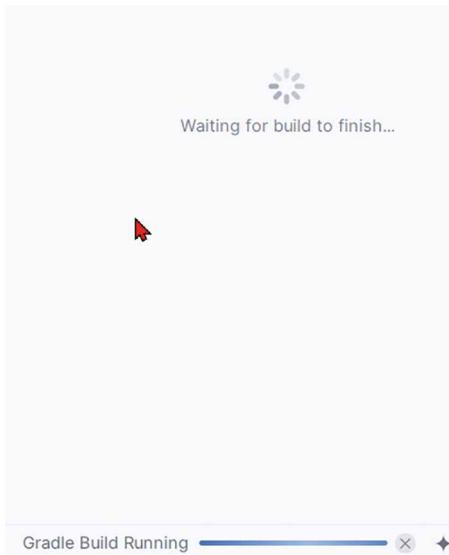


Figura 161



Eventualmente pode ocorrer um erro por desatualização. Nesse caso, a tela de Build Output mostrará o erro e a sua solução. Clique em Update.



Figura 162

Assim que a atualização for instalada, refaça a construção do Gradle clicando em Do Refactor.

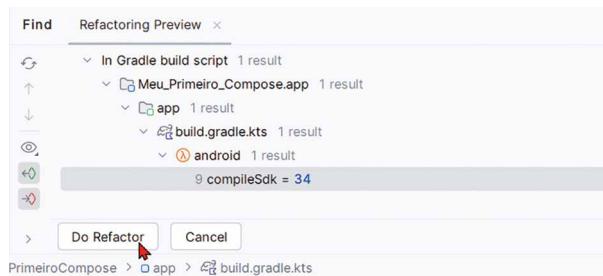


Figura 163

Unidade II

Aguarde a ressincronização do Gradle.

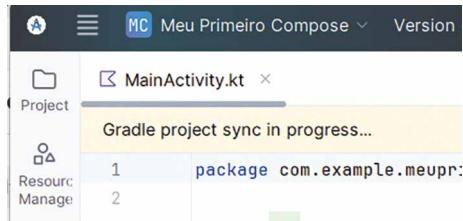


Figura 164

Assim que a sincronização terminar, refaça o build.

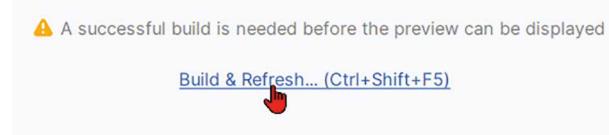


Figura 165

A tela do layout estará montada.

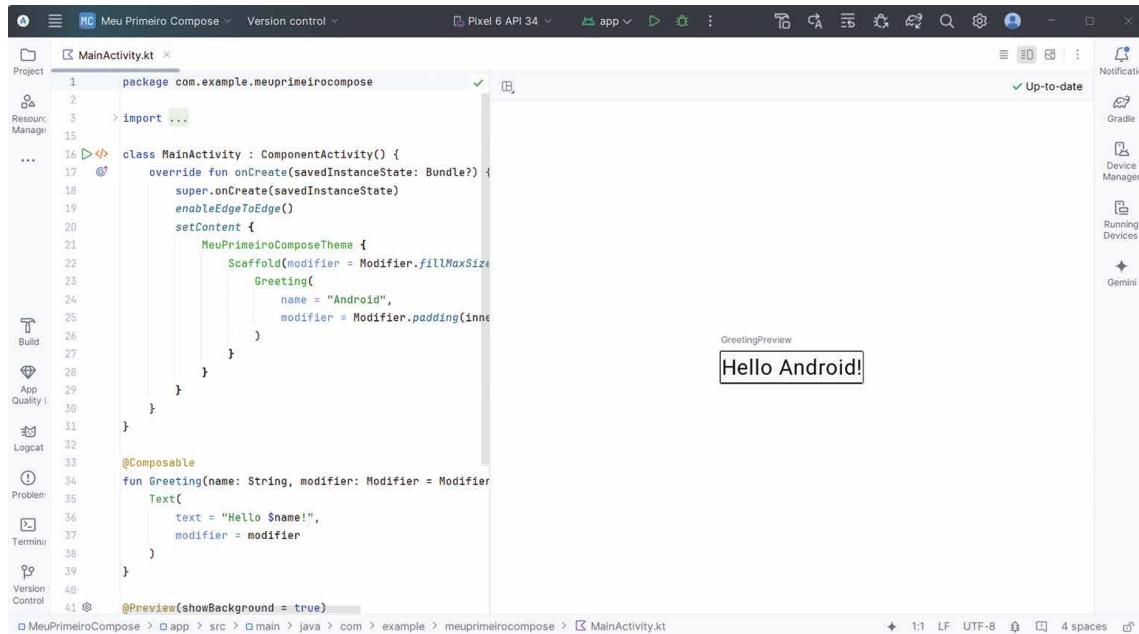


Figura 166

Inicie a execução do emulador. Problemas podem ocorrer, mas é importante lembrar que essa configuração inicial não possui erros de programação. Qualquer problema provavelmente é um conflito entre o hardware do computador e o emulador. Insista na execução e, se necessário, faça um cold boot na tela do Device Manager.



Figura 167

No código apresentado, temos:

```

class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            MeuExemploComposeLayout1Theme {
                Scaffold(
                    innerPadding -> Greeting(
                        "Android",
                        Modifier.padding(innerPadding)
                    )
                )
            }
        }
    }
}

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

@Preview(showBackground = true)
@Composable
fun GreetingPreview() {
    MeuExemploComposeLayout1Theme {           Função Preview
        Greeting("Android")
    }
}

```

Chamada da função

Função Composable

Função Preview

The diagram illustrates the structure of a composable function. It shows the code for the `MainActivity` class, which contains a `setContent` block. Inside this block, a `MeuExemploComposeLayout1Theme` is used to wrap a `Scaffold` component. The `Scaffold` component has a parameter `innerPadding` followed by a call to the `Greeting` composable function. The `Greeting` function takes two parameters: `name` (a string) and `modifier` (a `Modifier` object). Below this, there is a preview section for the `Greeting` function, showing it being called with the string "Android". A blue box labeled "Chamada da função" points to the call to the `Greeting` function in the `setContent` block. Two arrows point from the "Função Composable" box to the `Greeting` function definition and the "Função Preview" box to its definition.

Figura 168 – Estrutura do app composable

Explicando os comandos, temos:

- **setContent**: define o conteúdo da atividade usando uma função composable.
- **MeuExemploComposeLayout1Theme**: aplica um tema personalizado ao conteúdo.
- **Scaffold**: componente de layout que fornece uma estrutura básica para a tela, incluindo suporte para barras de ferramentas, navegação inferior etc. Aqui, ele usa o modificador Modifier.fillMaxSize() para ocupar todo o espaço disponível.
- **innerPadding -> Greeting(...)**: dentro do Scaffold, a função greeting é chamada, passando o nome Android, aplicando um padding interno e usando o modificador Modifier.padding(innerPadding).
- **Função Greeting**: função composable (por causa da anotação @Composable) que exibe um texto de saudação. Ela recebe um nome e um modificador como parâmetros e usa o elemento text para exibir a mensagem Hello, Android!

6.4.1 Introdução ao Jetpack Compose

O Jetpack Compose, um conjunto de ferramentas (framework) declarativo para a criação de interfaces de usuário no Android, teve um desenvolvimento bastante acelerado. Surgido como um projeto experimental, o compose rapidamente ganhou destaque pela sua promessa de simplificar e agilizar o desenvolvimento de UIs. Após um período de prévias e versões alfa, o framework atingiu a maturidade suficiente para ser integrado ao Android Studio, tornando-se uma ferramenta oficial para a criação de apps Android. Essa transição representou um marco importante, solidificando a posição do compose como o futuro da criação de interfaces no Android.

O paradigma de programação declarativa

Esse paradigma, ao contrário da programação imperativa, é um estilo de programação em que se descreve "o que" o programa deve fazer, em vez de como ele deve fazer. Em vez de especificar uma sequência de passos para alcançar um resultado, definem-se as propriedades e o estado desejado, e o sistema se encarrega de encontrar a melhor maneira de atingir esse estado. Imagine que você está pedindo para um cozinheiro preparar um prato. Na programação imperativa, você daria instruções detalhadas: pegue a panela, coloque óleo, adicione cebola, refogue por cinco minutos. Na programação declarativa, você simplesmente diria: eu quero um risoto de cogumelos. O cozinheiro sabe o que fazer para chegar ao resultado desejado.

Resumindo:

- **Imperativo**: descreve passo a passo como o programa deve executar uma tarefa. É como dar um algoritmo detalhado para o computador seguir.
- **Declarativo**: descreve o resultado final que deseja. O sistema se encarrega de encontrar a melhor maneira de alcançá-lo.

A programação declarativa no Compose é uma abordagem em que se descreve o que a interface de usuário deve ser, em vez de como construí-la passo a passo. Em vez de manipular diretamente os elementos da interface, define-se o estado desejado da tela e o compose se encarrega de atualizar a interface automaticamente conforme o estado muda. Isso simplifica o desenvolvimento, tornando o código mais intuitivo e menos propenso a erros, já que o framework gerencia as atualizações de forma eficiente.

6.4.2 Material Design

Material Design é um sistema de design criado pelo Google. Unifica e padroniza a experiência do usuário em diversos dispositivos e plataformas, oferecendo diretrizes, componentes e ferramentas para a criação de interfaces de usuário. O objetivo é garantir que os produtos digitais sejam intuitivos e consistentes, proporcionando uma experiência de usuário excelente (Google, 2024).

Utiliza princípios como física do mundo real, tipografia clara e layouts responsivos. O Material Design facilita a criação de interfaces amigáveis. Além disso, sua adaptabilidade permite que designers e desenvolvedores colaborem de forma eficaz na construção de produtos de alta qualidade.

O Jetpack Compose e o Material Design formam uma combinação poderosa para o desenvolvimento de aplicativos Android. Ao aproveitar a natureza declarativa do Compose e os componentes pré-construídos do Material Design, os desenvolvedores criam interfaces visuais funcionais, seguindo as diretrizes de design.

Os pilares do Material Design são:

- **Metáforas do mundo físico:** inspirado em materiais reais como papel, tinta e sombras, o Material Design cria interfaces com profundidade e dimensão.
- **Grids e espaçamento:** a utilização de grids e sistemas de espaçamento bem definidos assegura uma organização visual consistente e agradável.
- **Tipografia:** as fontes selecionadas são legíveis e versáteis, proporcionando uma experiência de leitura agradável.
- **Cores:** a paleta de cores é vibrante e expressiva, mantendo um equilíbrio harmonioso.
- **Animações:** animações suaves e intuitivas dão vida às interfaces, tornando a interação com o usuário agradável e natural.

Aqui estão os componentes do Material Design suportados pelo compose:

- Componentes básicos:

- **Text:** exibe texto.

- **Image**: exibe imagens.
- **Icon**: exibe ícones.
- **Box**: contêiner de layout flexível.
- **Surface**: cria uma superfície elevada com sombra.
- Layouts:
 - **Column**: organiza elementos verticalmente.
 - **Row**: organiza elementos horizontalmente.
 - **LazyColumn**: lista verticalmente rolável.
 - **LazyRow**: lista horizontalmente rolável.
 - **Grid**: organiza elementos em uma grade.
- Componentes de interação:
 - **Button**: botão clicável.
 - **TextField**: campo de texto para entrada de dados.
 - **Checkbox**: caixa de seleção.
 - **RadioButton**: botão de opção.
 - **Slider**: seleção de valores numéricos.
 - **Switch**: botão de alternância.
- Componentes de navegação:
 - **BottomNavigation**: barra de navegação inferior.
 - **NavigationRail**: barra de navegação lateral.
 - **TopAppBar**: barra de aplicativos superior.

- Componentes de diálogo e alerta:
 - **AlertDialog**: caixa de diálogo para mensagens importantes.
 - **Snackbar**: mensagem curta que aparece na parte inferior da tela.
 - **DropdownMenu**: menu suspenso.
- Componentes de progresso:
 - **CircularProgressIndicator**: indicador de progresso circular.
 - **LinearProgressIndicator**: indicador de progresso linear.
- Outros componentes:
 - **Divider**: separador visual.
 - **Spacer**: espaçador para criar espaço entre elementos.
 - **Spacer(Modifier.height(8.dp)) //**: exemplo de uso.
 - **DropdownMenuItem**: item de menu suspenso.
 - **DropdownMenu**.
 - **ExposedDropdownMenuBox**: menu suspenso com um campo de texto.
 - **OutlinedTextField**: campo de texto com contorno.
 - **Text composable**.

6.4.3 Widgets

Widgets compostáveis são funções em Jetpack Compose que criam interfaces de usuário de forma declarativa. Ao contrário dos métodos tradicionais de construção de UI, em que se manipula diretamente os elementos da interface, em Jetpack Compose, descreve como a interface deve ser em termos de funções compostáveis. Essas funções são anotadas com `@Composable` e podem ser combinadas para criar interfaces complexas e reutilizáveis.

Os parâmetros de um widget composable são usados para configurar e personalizar o comportamento e a aparência do widget. Eles são passados como argumentos para a função composable e incluem propriedades como texto, cores, tamanhos, ações de clique, entre outros.

O que fica entre as chaves {} em um widget composable é o conteúdo ou a lógica que define o comportamento interno do widget. Isso inclui outros widgets composable, funções lambda ou qualquer código que precise ser executado para construir a interface do usuário.

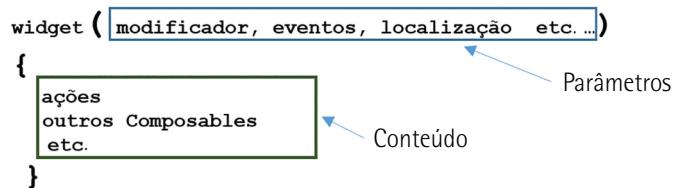


Figura 169

Ou para uma melhor visualização:

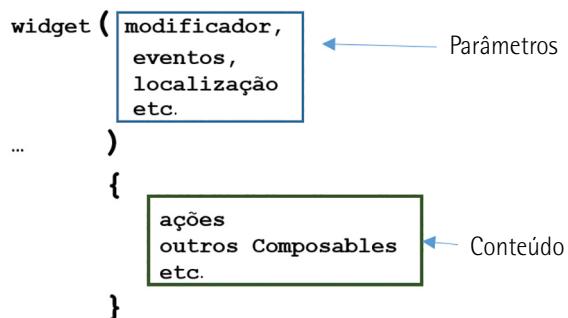


Figura 170

Exemplo:

Vamos analisar o seguinte layout:

```
Card(  
    modifier = Modifier.padding(16.dp),  
    elevation = CardDefaults.cardElevation(defaultElevation = 8.dp)  
) {  
    Column {  
        Text(text = "Título do Card",  
            style = MaterialTheme.typography.titleLarge)  
        Text(text = "Conteúdo do Card")  
        Button(onClick = { /* Lógica do botão */ }) {  
            Text(text = "Clique aqui")  
        }  
    }  
}
```

Figura 171

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Há uma série de widgets montados. Em verde, estão os widgets; em azul, os parâmetros, e dentro da caixa laranja os conteúdos.

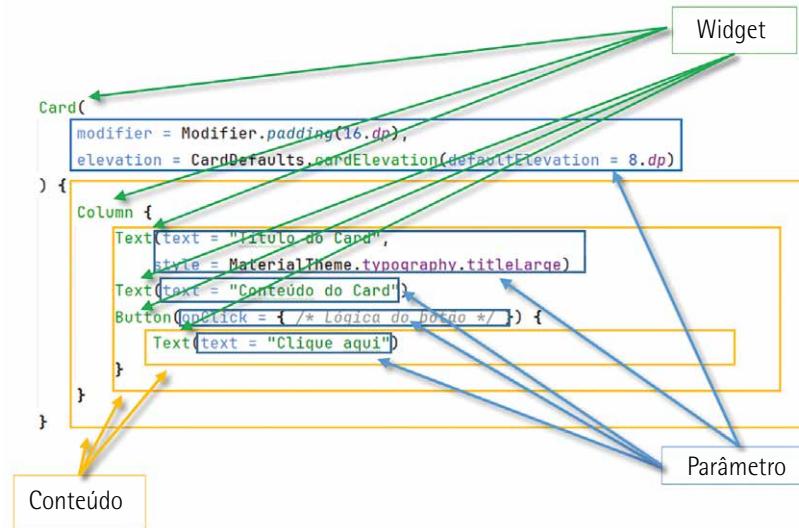


Figura 172 – Estrutura de um widget

No exemplo, temos:

Card é o widget principal, que representa um cartão. Modifier é um parâmetro que modifica o cartão, como padding. Elevation é outro parâmetro que define a elevação do cartão, criando uma sombra.

O conteúdo entre as chaves de card é um column que organiza o título e o conteúdo do cartão. O primeiro text exibe título do card e usa o estilo diferente do padrão. O segundo text exibe conteúdo do card.

O button adiciona um botão clicável dentro do card. O onClick define a ação a ser executada quando o botão é clicado (neste caso, está vazio e pode ser preenchido com a lógica desejada). Dentro do botão, há um text com o conteúdo clique aqui.

Analizando o resultado, temos:

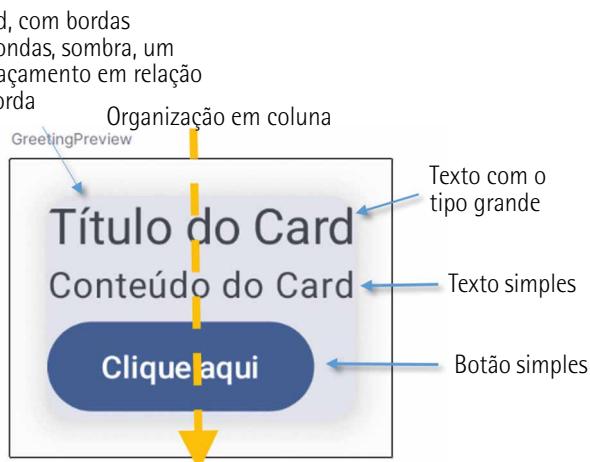


Figura 173

6.4.4 Fundamentos dos layouts

O Jetpack Compose proporciona uma abordagem flexível para criar interfaces de usuário declarativas no Android. Um dos pilares desse framework são os layouts, que organizam os elementos visuais na tela de maneira eficiente e adaptável. Essencialmente, ele se baseia em três conceitos principais:

- **Composables**: descrevem a UI. Cada composable retorna um elemento visual e combina com outros compostables para formar estruturas mais complexas.
- **Layout**: um composable que organiza outros compostables em uma estrutura visual. O Compose oferece vários layouts predefinidos, como column, row, box, lazycolumn, entre outros, cada um com suas características e finalidades.
- **Modificadores**: alteram o comportamento de um composable. Por exemplo, modificadores podem ser usados para definir alinhamento, padding, margin e outras propriedades visuais de um elemento.

O processo de criação de interfaces no Compose envolve três etapas principais: a composição de elementos individuais da UI, a organização desses elementos em estruturas visuais por layouts e a renderização desses elementos na tela, transformando o estado do aplicativo em uma interface visual interativa.

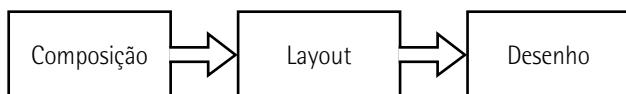


Figura 174

6.4.5 Composables

Funções compostáveis

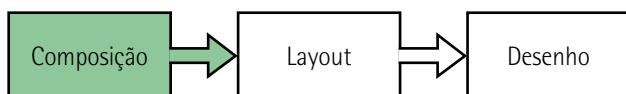


Figura 175

No Android Studio Compose, as funções compostáveis são fundamentais para construir interfaces de usuário. Em vez de criar layouts hierárquicos com XML, define-se a estrutura da UI de forma declarativa dentro dessas funções. Pense nelas como blocos de construção que podem ser combinados para criar interfaces complexas. Com o Compose, cria-se a interface do usuário definindo um conjunto de funções compostas que recebem dados e geram elementos da IU. Um exemplo simples é o widget greeting presente no programa modelo, que recebe uma string e um modificador e gera um widget text, exibindo uma mensagem de saudação.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

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

Alguns detalhes importantes sobre essa função:

- A função é anotada com a anotação @Composable. Todas as funções que podem ser compostas precisam ter essa anotação. Ela informa ao compilador do Compose que a função é usada para converter dados em UI.
- A função recebe dados. As funções que podem ser compostas aceitam parâmetros, o que permite que a lógica do app descreva a UI. Nesse caso, o widget aceita uma string para cumprimentar o usuário pelo nome e um parâmetro opcional do tipo modifier, que tem um valor padrão de modifier.
- A função exibe texto na UI. Isso é feito chamando a função combinável Text(), que cria o elemento da UI de texto. Essas funções que podem ser compostas emitem a hierarquia da UI chamando outras funções desse tipo. O modifier = modifier. O parâmetro modifier é passado para o text, permitindo que a aparência e o comportamento do text sejam personalizados externamente.
- A função não retorna nada. As funções do Compose que emitem a UI não precisam retornar nada, porque descrevem o estado desejado da tela em vez de construir widgets de UI.

6.4.6 Principais layouts em Compose

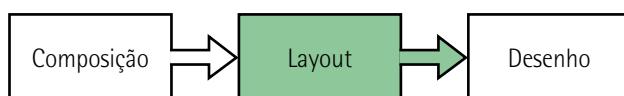


Figura 176

Para compreender como as funções compostas funcionam, analisaremos o exemplo do Android Studio. Inicialmente temos o dispositivo vazio:

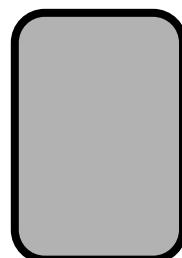


Figura 177

No MainActivity.kt, no evento onCreate dispositivo, solicitamos para iniciar a configuração do conteúdo da activity usando o Jetpack Compose.

```
setContent { ... }
```

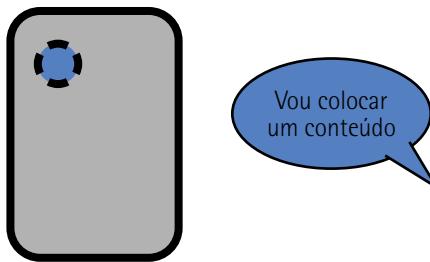


Figura 178

O estilo visual da interface chamado MeuExemploComposeLayout1Theme é definido para o conteúdo.

```
MeuExemploComposeLayout1Theme {
```

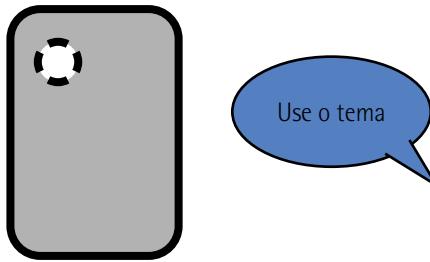


Figura 179

O componente Scaffold é utilizado para fornecer uma estrutura básica para a tela. Ele usa o modificador Modifier.fillMaxSize() para ocupar todo o espaço disponível na tela.

```
Scaffold(modifier = Modifier.fillMaxSize()) {
```



Figura 180

Dentro do scaffold, há a função greeting. Ela recebe o nome Android como parâmetro e aplica um padding interno usando o modificador Modifier.padding(innerPadding), que ajusta o espaçamento interno com base no layout do scaffold.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
innerPadding -> Greeting(  
    name = "Android",  
    modifier = Modifier.padding(innerPadding)  
)
```

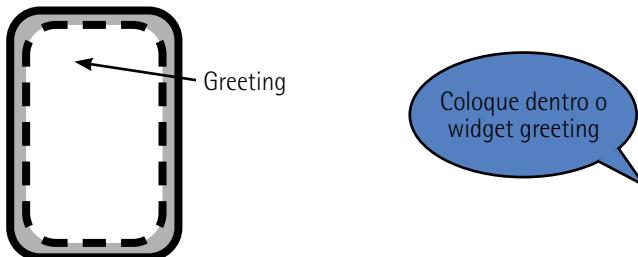


Figura 181

Assim, o contorno da activity está definido e esperando continuar na função (widget) greeting:

```
setContent {  
    MeuExemploComposeLayout1Theme {  
        Scaffold(modifier = Modifier.fillMaxSize()) {  
            innerPadding -> Greeting(  
                name = "Android",  
                modifier = Modifier.padding(innerPadding)  
)  
        }  
    }  
}
```

No greeting, temos:

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

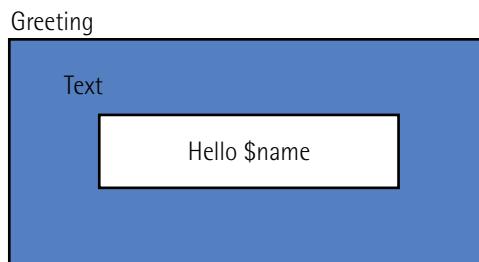


Figura 182

Unidade II

Então há a concretização do processo de criação:



Figura 183

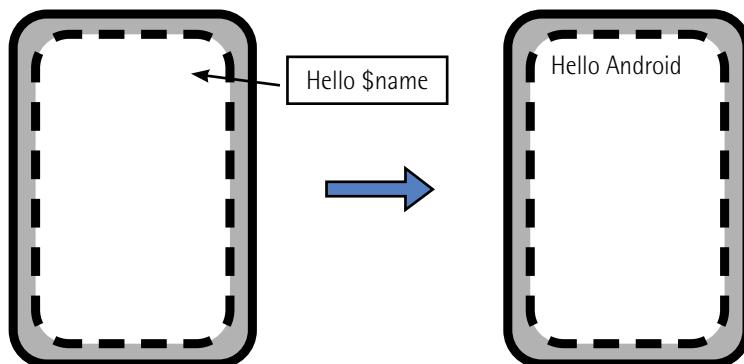


Figura 184

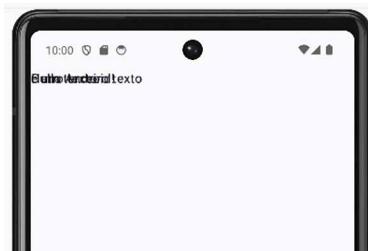
Alteraremos a função greeting desta vez colocando mais dois text.

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

Veja o resultado:



Preview



Emulador

Figura 185

Os textos estão todos sobrepostos. Uma função composable pode emitir vários elementos da interface. No entanto, se não especificar como eles devem ser organizados, o Compose pode dispor os elementos de forma indesejada. Para resolver, utilizaremos os layouts a seguir.

Column

O column é um dos componentes essenciais do Jetpack Compose para criar layouts verticais. Ele atua como um contêiner que organiza seus elementos-filhos em uma pilha vertical, um abaixo do outro. Essa estrutura facilita a criação de interfaces que seguem o padrão de leitura de cima para baixo, comum em muitas culturas. Além de sua função básica de empilhar elementos, o column oferece várias opções de personalização, como alinhamento dos elementos, espaçamento entre eles e a possibilidade de aplicar modificadores para customizar a aparência e o comportamento de cada item. Com o column, os desenvolvedores podem construir interfaces intuitivas e eficientes de forma declarativa e concisa.

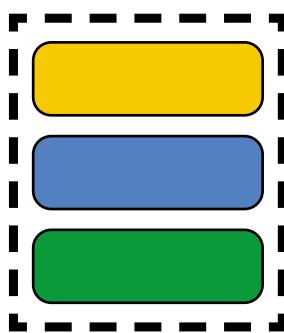


Figura 186 – Layout column

Modificaremos a função para corrigir a sobreposição dos textos, envolvendo os widgets com o layout column.

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Column {
        Text(
            text = "Hello $name!",
            modifier = modifier
        )
        Text(
            text = "Outro texto",
            modifier = modifier
        )
        Text(
            text = "e um terceiro texto",
            modifier = modifier
        )
    }
}
```

Figura 187

Assim os textos não estão mais sobrepostos:

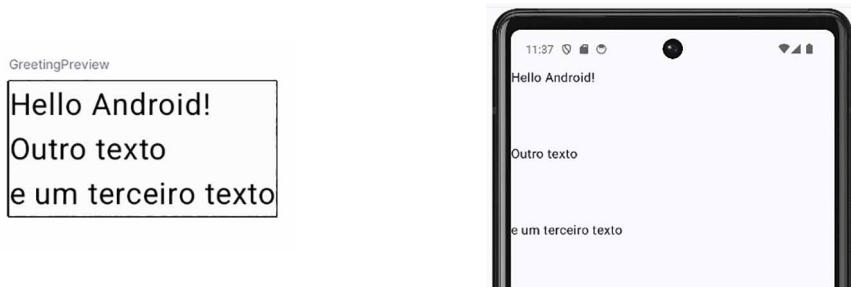


Figura 188

Use os modificadores para ajustar o comportamento e a aparência dos elementos dentro do column, definindo propriedades como padding, margem e alinhamento. O column também permite configurar o alinhamento dos seus filhos, posicionando os elementos no início, centro ou fim da coluna, tanto horizontal quanto verticalmente.

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Column (
        modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text(
            text = "Hello $name!",
            modifier = modifier
        )
        Text(
            text = "Outro texto",
            modifier = modifier
        )
        Text(
            text = "e um terceiro texto",
            modifier = modifier
        )
    }
}
```

Figura 189

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Os textos agora estão no centro da activity.

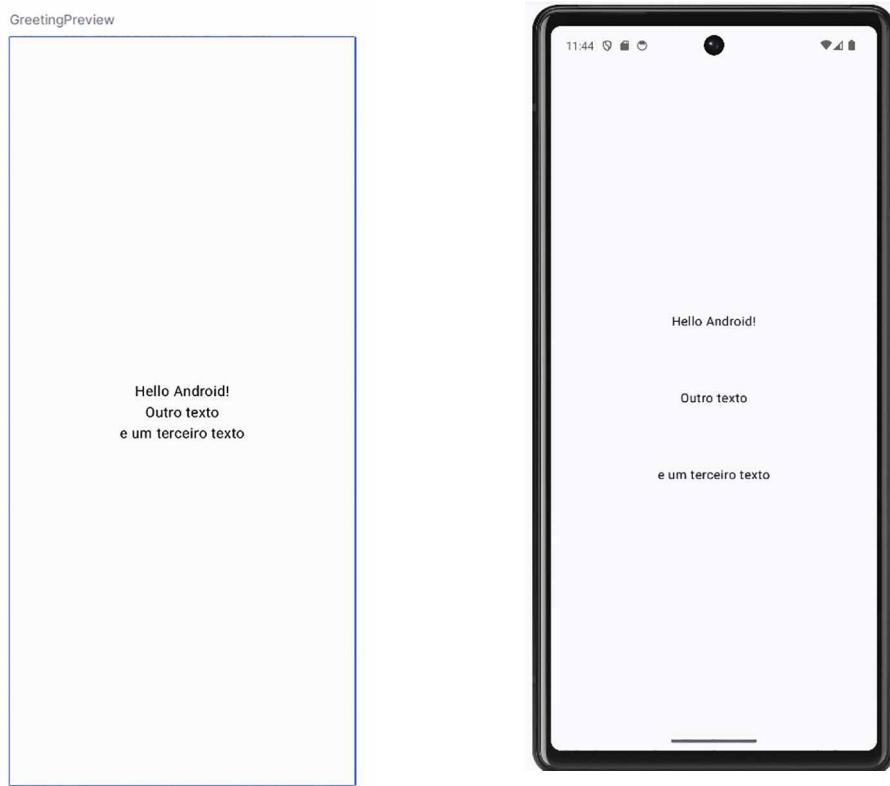


Figura 190

Se você precisar de uma coluna que possa ser rolada, ou seja, que permita a navegação vertical por meio de uma lista de itens, utilize o modificador vertical scroll.

```
Column (
    modifier = Modifier.verticalScroll(rememberScrollState())
) {
    Text(
        text = "Hello $name!",
        modifier = modifier
    )
    Text(
        text = "Outro texto",
        modifier = modifier
    )
    Text(
        text = "e um terceiro texto",
        modifier = modifier
    )
}
```

Figura 191

Row

O row no Jetpack Compose é um componente para criar layouts horizontais. Ele funciona como um contêiner que organiza seus elementos-filhos em uma linha, um ao lado do outro. Essa estrutura é ideal para criar interfaces que seguem o padrão de leitura horizontal, como botões alinhados lado a lado ou informações dispostas em colunas.

Assim como o column, o row oferece diversas opções de personalização, como alinhamento dos elementos, espaçamento entre eles e a possibilidade de aplicar modificadores para customizar a aparência e o comportamento de cada item. Com o row, os desenvolvedores podem construir interfaces flexíveis e adaptáveis, em diferentes tamanhos de tela.



Figura 192 – Layout row

No exemplo do column, vamos alterar o layout para row.

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Row {
        Text(
            text = "Hello $name!",
            modifier = modifier
        )
        Text(
            text = "Outro texto",
            modifier = modifier
        )
        Text(
            text = "e um terceiro texto",
            modifier = modifier
        )
    }
}
```

Figura 193

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

A tela resultante fica:

```
GreetingPreview
Hello Android!Outro texto e um terceiro texto
```



Figura 194

Assim com columns e rows, fazemos associações:

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Row {
        Text(
            text = "Hello $name!",
            modifier = modifier
        )
        Column {
            Text(
                text = "Outro texto",
                modifier = modifier
            )
            Text(
                text = "e um terceiro texto",
                modifier = modifier
            )
        }
    }
}
```

Figura 195

Temos o resultado:

```
GreetingPreview
Hello Android!Outro texto
e um terceiro texto
```



Figura 196

Unidade II

Podemos alterar também o alinhamento:

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

Figura 197

Utilizando o alinhamento CenterVertically, os conteúdos ficam alinhados no meio da altura.



Figura 198

Exemplo: neste momento é possível compreender um pouco mais o funcionamento do Compose. Criaremos uma função @Composable chamada linha exatamente igual ao greeting.

```
@Composable
fun linha(name: String, modifier: Modifier = Modifier) {
    Row(
        verticalAlignment = Alignment.CenterVertically
    ) {
        Text(
            text = "Hello $name!",
            modifier = modifier
        )
        Column {
            Text(
                text = "Outro texto",
                modifier = modifier
            )
            Text(
                text = "e um terceiro texto",
                modifier = modifier
            )
        }
    }
}
```

Figura 199

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Apagamos todo o conteúdo da função e colocamos a chamada para a nova função criada.

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

@Composable
fun linha(name: String, modifier: Modifier = Modifier) {
    Row(
        verticalAlignment = Alignment.CenterVertically
    ) {
        Text(
            text = "Hello $name!",
            modifier = modifier
        )
        Column {
            Text(
                text = "Outro texto",
                modifier = modifier
            )
            Text(
                text = "e um terceiro texto",
                modifier = modifier
            )
        }
    }
}
```

Figura 200

Observe que o resultado é o mesmo:

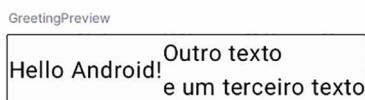


Figura 201

Simplesmente transferimos a montagem da uma nova função. Desta forma, o resultado é o mesmo.

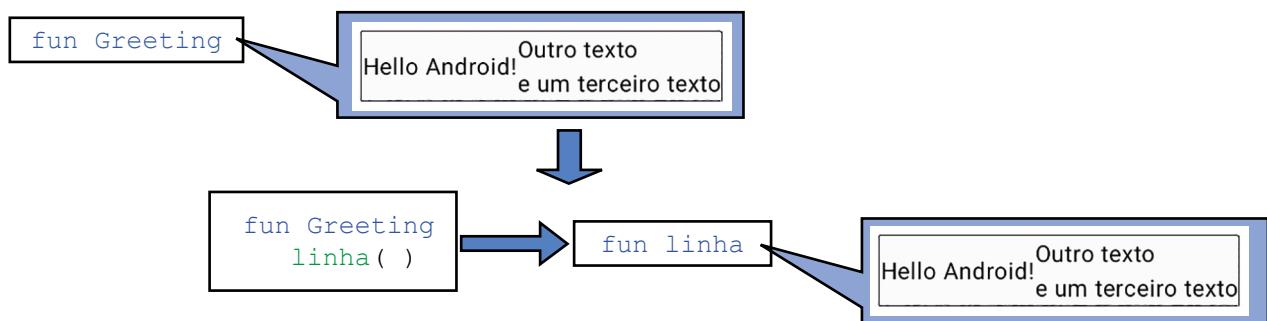


Figura 202

Unidade II

Na função greeting, colocamos um column e, dentro dele, três chamadas para a função linha.

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

Figura 203

Observe agora que o resultado é:

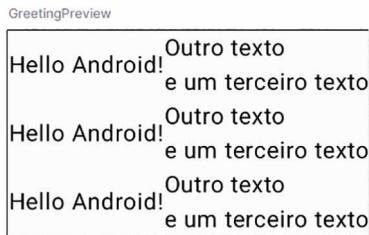


Figura 204

Como cada chamada da função linha está dentro de um layout column, cada um dos resultados aparece abaixo do outro.

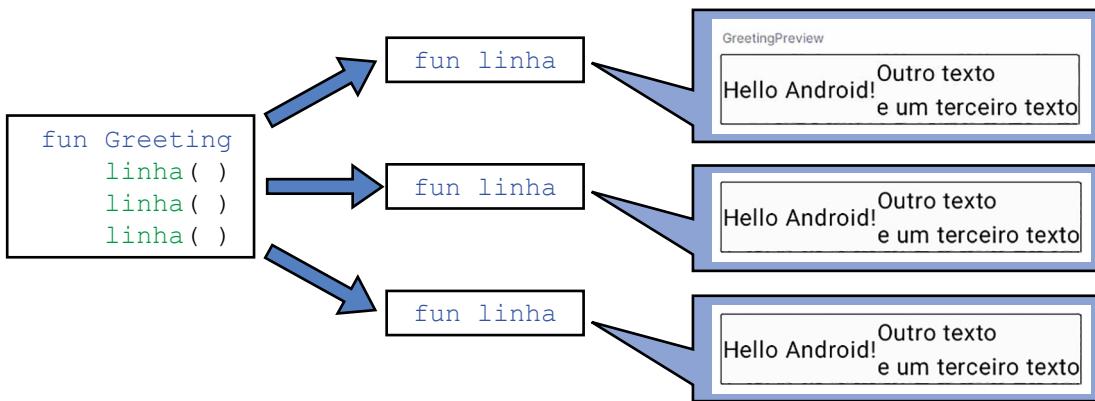


Figura 205

Para automatizar a exibição da função linha dez vezes, em vez de digitarmos a chamada dez vezes, usamos um laço de repetição. Isso fará a chamada da função linha de forma automática. Para diferenciar cada linha, adicionamos a informação do contador ao nome.

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Column {
        for (i in 1 .. 10) {
            linha(name = name+i.toString(), modifier = modifier)
        }
    }
}
```

Figura 206

Observação

O contador do laço é um número inteiro, enquanto o name é um caractere. Portanto, precisamos converter o tipo da variável usando o método `toString()`. Assim, se o conteúdo de name for Android e o valor da variável i for o número 1, ao usar o método `.toString()`, ele será transformado em 1, um caractere. Assim podemos juntar esses dois caracteres. Isso é chamado concatenação.

`name+i.toString()` resulta em "Android1" para `name="Android"` e `i = 1`

O resultado após o laço:

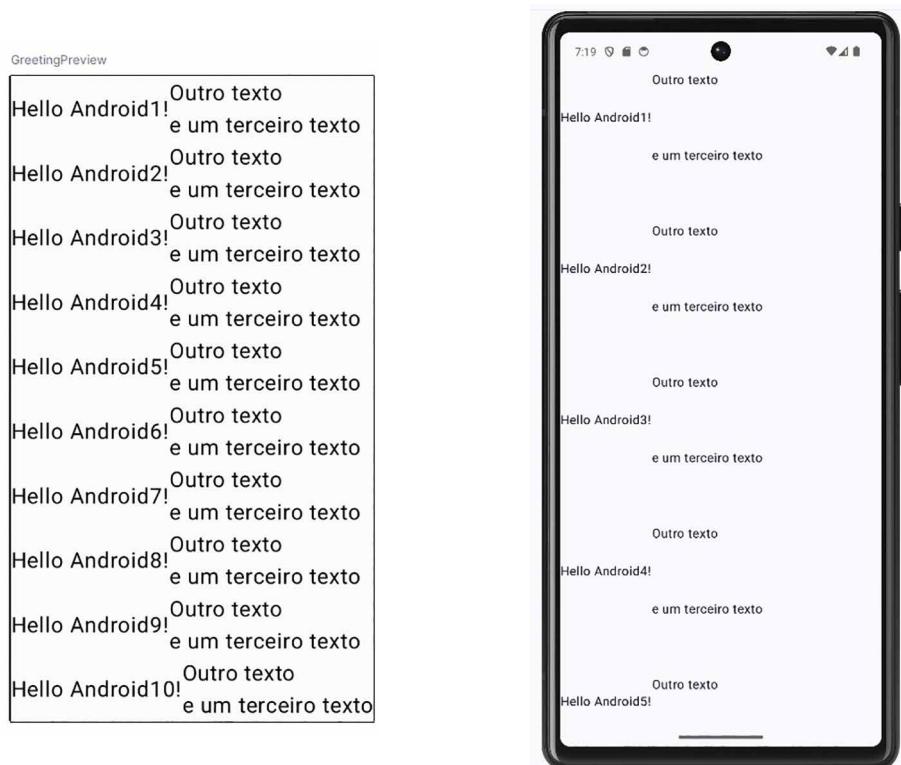


Figura 207



Lembrete

O laço de repetição foi visto em 2.1.4 Repetição.

Box

O box é um layout flexível que organiza seus elementos de forma livre e personalizada. Diferentemente do row e do column, que restringem os elementos a um eixo horizontal ou vertical, o box oferece um controle mais preciso sobre o posicionamento de cada elemento. Você pode definir margens, preenchimentos, alinhamentos e sobrepor elementos. Isso o torna ideal para criar layouts complexos e personalizados.

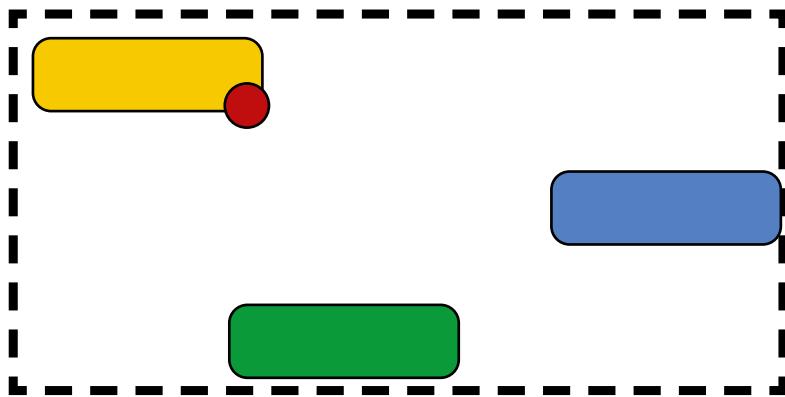


Figura 208 – Layout box

Modificaremos o conteúdo da função greeting para incluir um box que ocupará todo o espaço da tela. Esse box terá um fundo (background) cinza-claro (Color.LightGray) e um preenchimento (padding) de 32 pixels. Dentro dele, colocaremos três textos.

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Box(
        modifier = Modifier.fillMaxSize()
            .padding(32.dp)
            .background(Color.LightGray)
    ) {
        Text(text = "Texto superior esquerdo",
            modifier = Modifier.align(Alignment.TopStart))
        Text(text = "Texto centralizado",
            modifier = Modifier.align(Alignment.Center))
        Text(text = "Texto inferior direito",
            modifier = Modifier.align(Alignment.BottomEnd))
    }
}
```

Figura 209

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

O primeiro texto ficará no início do topo do layout, o segundo no centro e o terceiro no fim da última linha.

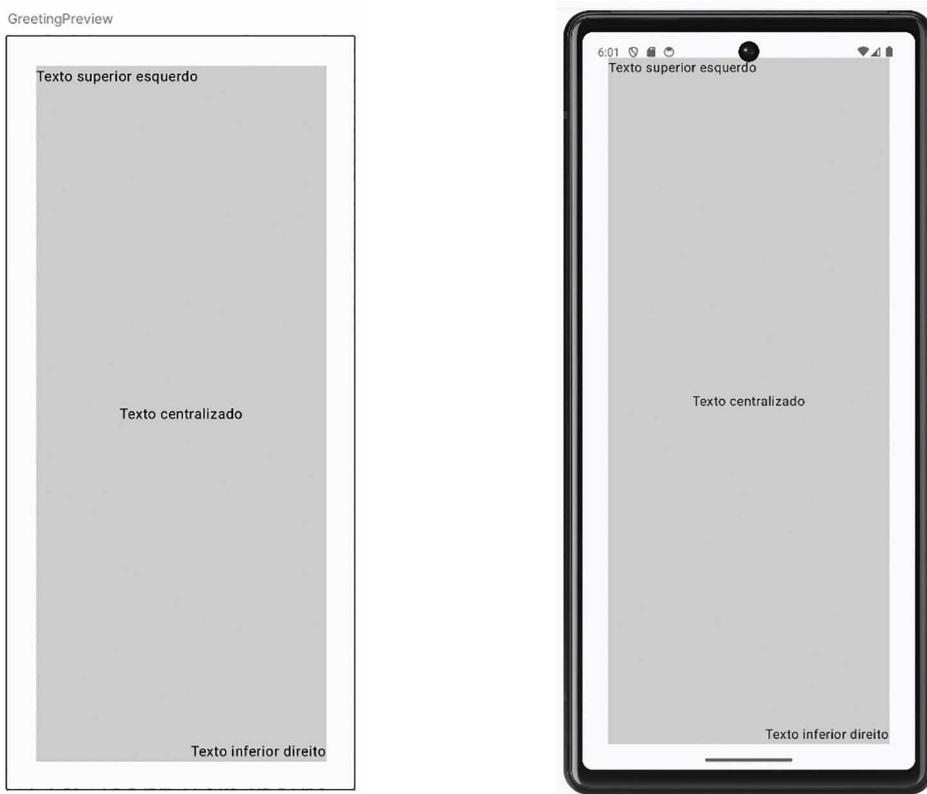


Figura 210

LazyColumn e LazyRow

Os componentes LazyColumn e LazyRow são otimizados para renderizar listas longas de itens de forma eficiente. Ao contrário do column e row tradicionais, que renderizam todos os itens de uma vez, os LazyColumn e LazyRow somente renderizam os itens que estão atualmente visíveis na tela. Essa abordagem é crucial para evitar problemas de desempenho, especialmente em listas com muitos itens.

O LazyColumn cria uma lista que rola verticalmente, enquanto o LazyRow cria uma lista que rola horizontalmente. Por exemplo, para criar uma lista vertical de itens, use o LazyColumn e dentro dele itere sobre uma lista de dados, criando um item visual para cada elemento da lista. Essa abordagem é ideal para implementar listas de mensagens, feeds de notícias ou qualquer outra lista longa que precise ser rolável.

Para esse exemplo, modificaremos completamente a função greeting. Criaremos uma lista numerada com cem itens e iteraremos sobre cada linha dessa lista para exibi-la na tela. Como as cem linhas não cabem na tela do dispositivo, utilizaremos o LazyColumn para permitir a rolagem vertical, possibilitando a navegação para cima e para baixo (figura 211).

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    //
    val listaNumeros: MutableList<String> = mutableListOf()
    // Adiciona elementos à lista
    for (i in 1 .. 100) {
        listaNumeros.add("numero $i")
    }
    LazyColumn(
        modifier = Modifier.fillMaxSize(),
        contentPadding = PaddingValues(16.dp),
        verticalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        items(listaNumeros) { listaNumeros ->
            Text(
                text = "Hello, ${listaNumeros}",
                modifier = modifier
            )
        }
    }
}
```

Figura 211

Entendendo esse trecho, criamos uma lista vazia:

```
val listaNumeros: MutableList<String> = mutableListOf()
```

Preenchemos a lista com número 1, número 2, número 3 e assim por diante, até número 100.

```
// Adiciona elementos à lista
for (i in 1..100) {
    listaNumeros.add("numero $i")
}
```

A configuração do LazyColumn apresentada define uma lista vertical que ocupa todo o espaço disponível na tela, graças ao modificador Modifier.fillMaxSize(). O parâmetro contentPadding = PaddingValues(16.dp) adiciona um preenchimento de 16 dp ao redor do conteúdo, garantindo que os itens não fiquem encostados nas bordas da tela. Além disso, o verticalArrangement = Arrangement.spacedBy(8.dp) define um espaçamento vertical de 8 dp entre cada item da lista, proporcionando uma aparência organizada e espaçada.

```
LazyColumn(
    modifier = Modifier.fillMaxSize(),
    contentPadding = PaddingValues(16.dp),
    verticalArrangement = Arrangement.spacedBy(8.dp)
) { }
```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Internamente ao LazyColumn, temos:

```
items(listaNumeros) { listaNumeros ->
    Text(
        text = "Hello, $listaNumeros",
        modifier = modifier
    )
}
```

Essa expressão é usada para iterar sobre uma lista chamada listaNumeros e exibir cada elemento como um texto. Vamos detalhar:

- **items(listaNumeros)**: recebe a lista listaNumeros e itera sobre cada elemento dela.
- **{ listaNumeros -> ... }**: lambda que define o que fazer com cada elemento da lista. Aqui, listaNumeros é o nome do parâmetro que representa cada item da lista durante a iteração.
- **Text(...)**: dentro do lambda, a função text exibe cada elemento da lista como um texto.

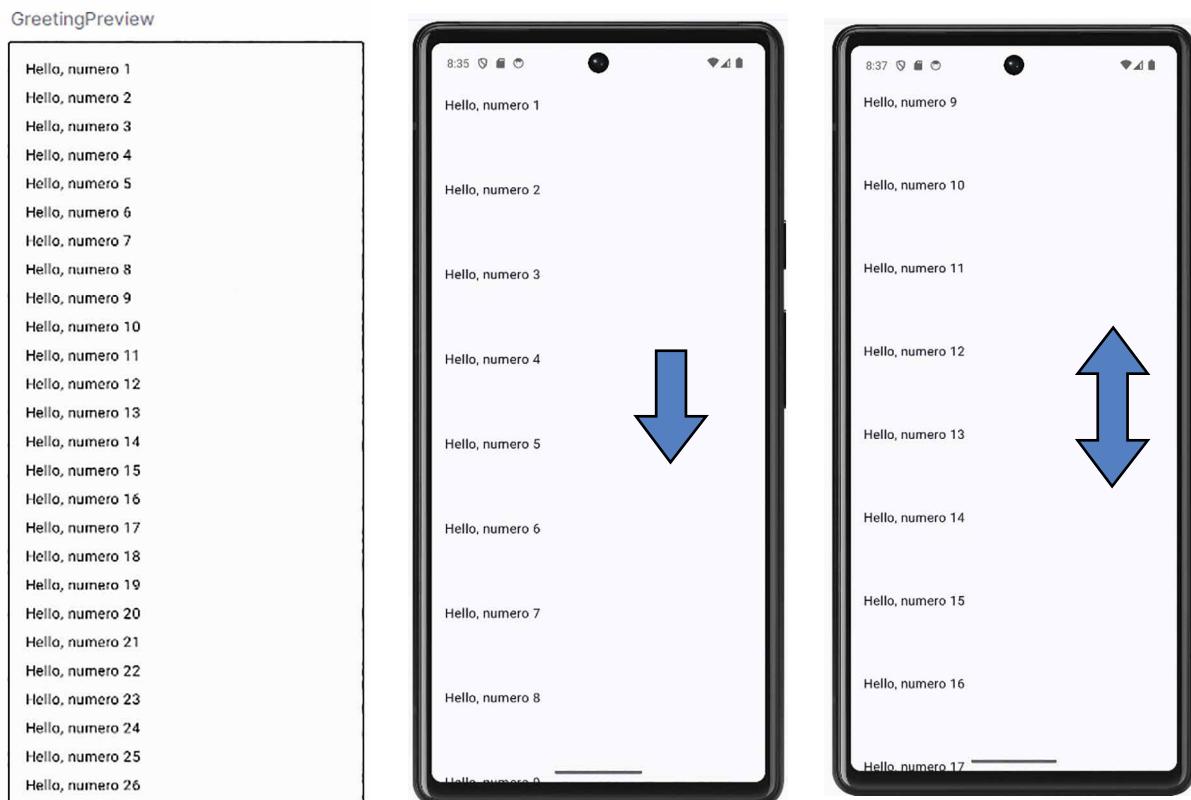


Figura 212



Lembrete

Na figura 212, utilizamos uma função lambda, cuja comprehensão é essencial, pois ela é amplamente utilizada na programação em Kotlin.

Grid

No Compose, os layouts de grade são utilizados para organizar itens em uma estrutura de linhas e colunas, proporcionando uma maneira eficiente de exibir coleções de dados. O LazyVerticalGrid é um dos componentes mais comuns para esse propósito, permitindo que os itens sejam dispostos em uma grade verticalmente rolável. Isso é útil para listas grandes ou de comprimento desconhecido, pois apenas os itens visíveis são compostos e renderizados, melhorando o desempenho.

Alteraremos a função greeting para visualizar o grid.

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    LazyVerticalGrid (
        columns = GridCells.Fixed(3)
    ) {
        items(6) { indice ->
            Text(
                text = "Célula $indice",
                modifier = Modifier.background(Color.LightGray)
            )
        }
    }
}
```

Figura 213

Analisando o novo conteúdo da função:

```
LazyVerticalGrid (
    columns = GridCells.Fixed(3),
    modifier = modifier
)
```

Cria-se uma grade vertical com três colunas fixas. O parâmetro modifier é aplicado ao grid, recebendo a predefinição do espaço da tela que foi enviado pela chamada na função principal.

```
items(6) { índice ->
```

Define-se que a grade terá seis itens.

```
{ índice ->
...
}
```

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

O lambda é executado para cada item, em que o índice é do item atual.

```
Text(  
    text = "Célula $indice",  
    modifier = Modifier.background(Color.LightGray)  
)
```

Para cada item, um componente text é criado, exibindo o texto célula seguido do índice do item. O modificador Modifier.background(Color.LightGray) define um fundo cinza-claro para cada célula.

Assim, tem-se o resultado:



Figura 214

Surface

O surface é um componente que representa uma superfície material, aplicando estilos de Material Design. Ele é utilizado para criar elementos visuais com profundidade e hierarquia em sua interface. O surface oferece propriedades como elevação, sombra, cor e forma, permitindo a construção de componentes com aspecto mais realista e tridimensional. Além disso, o surface se adapta de modo automático ao tema do sistema, garantindo uma experiência visual consistente com o resto do aplicativo.

No exemplo, usaremos o surface para criar um contêiner estilizado que envolve um texto. Utilizaremos o surface com várias propriedades estilísticas: cor de fundo azul (color = Color.Blue), cantos arredondados com um raio de 8 dp (shape = RoundedCornerShape(8.dp)), elevação da sombra de 3 dp (shadowElevation = 3.dp), cor do conteúdo branca (contentColor = Color.White) e uma borda preta de 2 dp de espessura (border = BorderStroke(2.dp, Color.Black)). Essas propriedades serão aplicadas ao componente text.

```
@Composable  
fun Greeting(name: String, modifier: Modifier = Modifier) {  
    Surface(  
        color = Color.Blue,  
        shape = RoundedCornerShape(8.dp),  
        shadowElevation = 3.dp,  
        contentColor = Color.White,  
        border = BorderStroke(2.dp, Color.Black),  
        modifier = modifier  
    ) {  
        Text(  
            text = "Hello $name!",  
            modifier = modifier  
        )  
    }  
}
```

Figura 215

Assim tem-se:



Figura 216

Scaffold

O scaffold no Jetpack Compose é um componente fundamental que estabelece uma estrutura padrão para a criação de telas em aplicativos. Ele funciona como um layout base, disponibilizando slots ou espaços reservados para inserir componentes específicos, como a barra superior (top bar), barra inferior (bottom bar), botão de ação flutuante (FAB) e o corpo principal da tela. Esses slots asseguram organização e consistência na estrutura das suas telas. Utilizar o scaffold simplifica a criação de interfaces de usuário mais complexas e integradas.

Saiba mais

Para conhecer mais o scaffold, acesse o site a seguir.

DEVELOPERS. *Scaffold*. [s.d.]h. Disponível em: <https://shre.ink/M0zr>. Acesso em: 6 mar. 2025.

O scaffold já estava em uso nos exemplos até aqui, mas sem explicações detalhadas.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            MeuExemploCompose6aTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "Android",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}
```

Figura 217

A tela inicial conforme já vimos é esta:



Figura 218

Até agora, a explicação para sua presença era a definição da estrutura da tela do aplicativo. Agora, veremos como essa estrutura é construída. Acrescentaremos uma barra padrão no topo da tela.

No momento da escrita deste livro-texto, um novo mecanismo está em teste, portanto, será necessário colocar uma anotação antes do onCreate:

```
@OptIn(ExperimentalMaterial3Api::class)
```

Acrescentaremos na chamada do scaffold a inclusão de uma barra azul no topo do aplicativo:

```
class MainActivity : ComponentActivity() {
    @OptIn(ExperimentalMaterial3Api::class)
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            MeuExemploCompose6aTheme {
                Scaffold(
                    topBar = {
                        TopAppBar(
                            title = { Text("Barra do Topo") },
                            colors = TopAppBarDefaults.topAppBarColors(Color.Blue)
                        )
                    },
                    modifier = Modifier.fillMaxSize()
                ) { innerPadding ->
                    Greeting(
                        name = "Android",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }
}
```

Figura 219

A estrutura básica da tela é definida usando o componente scaffold, que inclui uma barra superior (topBar). A barra superior é criada com o componente TopAppBar, que exibe o texto barra do topo e tem a cor de fundo azul, definida por TopAppBarDefaults.topAppBarColors(Color.Blue).

O scaffold também utiliza o modificador Modifier.fillMaxSize() para ocupar todo o espaço disponível na tela. Isso resulta em uma interface de usuário com uma barra superior azul e um título, proporcionando uma estrutura organizada.



Figura 220

Adicionaremos uma barra inferior (bottomBar) ao scaffold usando o componente BottomAppBar. A cor de fundo da barra inferior é definida como verde por meio do parâmetro containerColor = Color.Green. Dentro da BottomAppBar, há um texto com o conteúdo Barra Inferior, cuja cor é definida como azul (color = Color.Blue), conforme a figura 221.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
Scaffold(  
    topBar = {  
        TopAppBar(  
            title = { Text("Barra do Topo") },  
            colors = TopAppBarDefaults.topAppBarColors( Color.Blue )  
        )  
    },  
    bottomBar = { // Add bottomBar parameter  
        BottomAppBar(  
            containerColor = Color.Green  
        ) {  
            Text("Barra Inferior", color = Color.Blue)  
        }  
    },  
    modifier = Modifier.fillMaxSize()  
) { innerPadding ->
```

Figura 221

Resulta em uma barra inferior verde com um texto azul, proporcionando uma estrutura visualmente distinta na parte inferior da tela.



Figura 222

Adicionaremos um botão de ação flutuante (floatingActionButton) ao scaffold usando o componente FloatingActionButton. Quando o botão é clicado, a função definida em onClick é executada (neste caso, está vazia e pode ser preenchida com a ação desejada). Dentro do botão, há um ícone (icon) que utiliza o vetor de imagem padrão Icons.Default.Add, um símbolo de + e uma descrição de conteúdo Adicionar.

Unidade II

Isso resulta em um botão flutuante com um ícone de adição, proporcionando uma maneira fácil e acessível para os usuários executarem uma ação específica na interface do aplicativo.

```
Scaffold(  
    topBar = {  
        TopAppBar(  
            title = { Text("Barra do Topo") },  
            colors = TopAppBarDefaults.topAppBarColors( Color.Blue )  
        )  
    },  
    bottomBar = { // Add bottomBar parameter  
        BottomAppBar(  
            containerColor = Color.Green  
        ) {  
            Text("Barra Inferior", color = Color.Blue)  
        }  
    },  
    floatingActionButton = {  
        FloatingActionButton(onClick = { /* ação ao clicar */ }) {  
            Icon(imageVector = Icons.Default.Add,  
                  contentDescription = "Adicionar"  
            )  
        }  
    },  
    modifier = Modifier.fillMaxSize()  
) { innerPadding ->
```

Figura 223

Temos como resultado:

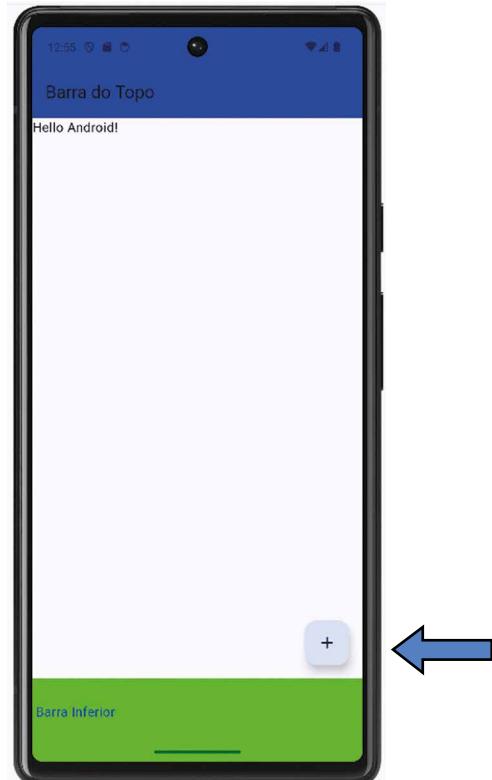


Figura 224

6.4.7 Modificadores de layout

Modificadores são objetos que ajustam o comportamento, a aparência ou o layout de composables sem alterar seu estado interno. Eles permitem personalizar propriedades como tamanho, preenchimento, alinhamento, cor de fundo, ações de clique e muito mais. Sendo imutáveis, os modificadores podem ser combinados (ou encadeados) para adicionar múltiplos efeitos a um único composable. Assim, eles permitem decorar ou aprimorar um composable.

Modificadores oferecem um controle sobre os componentes. Eles permitem:

- **Customizar a aparência:** alterar tamanho, forma, cor e outros atributos visuais.
- **Definir o comportamento:** tornar um elemento clicável, rolável ou arrastável.
- **Melhorar a acessibilidade:** adicionar rótulos e descrições para auxiliar usuários com deficiência.
- **Gerenciar o layout:** controlar o posicionamento e o tamanho dos elementos na tela.

Estrutura básica de um modificador

Os modificadores geralmente são aplicados usando o parâmetro modifier de um composable. Vamos adicionar ao modifier já vindo como parâmetro um valor para definir uma margem de 16 pontos.

```
Text(  
    text = "Hello $name!",  
    modifier = modifier.padding(16.dp)  
)
```

Temos o seguinte resultado:



Figura 225

Alterando o valor para 50.dp, temos:

```
modifier = modifier.padding( 50. dp)
```

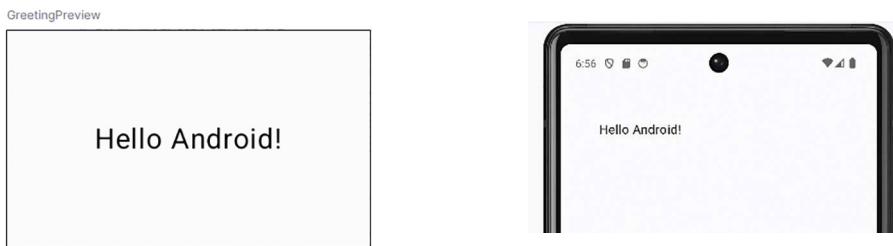


Figura 226

Modificadores de layout

Os modificadores de layout ajustam como um composable é posicionado dentro de seu layout-pai. Eles ajudam a gerenciar o tamanho, a posição e o alinhamento de um composable dentro do espaço disponível.

padding()

O modificador padding() adiciona preenchimento dentro do composable, criando espaço entre o conteúdo do composable e suas bordas.

```
Box( modifier = modifier
    .padding ( 16 .dp)
    .size ( 200 .dp)
    .background ( Color .Red)
) {  
  
    Text( text = "Hello $name!", color = Color.White, modifier = modifier
        .padding(10.dp)
        .background ( Color .Blue)
    )
}
```

Figura 227

O modificador padding(16.dp) adiciona 16 dp de preenchimento dentro do Box, afastando o conteúdo (Texto) das bordas da caixa e criando uma margem interna. Como o preenchimento é um modificador de layout, ele deve ser aplicado antes dos modificadores de tamanho e desenho.

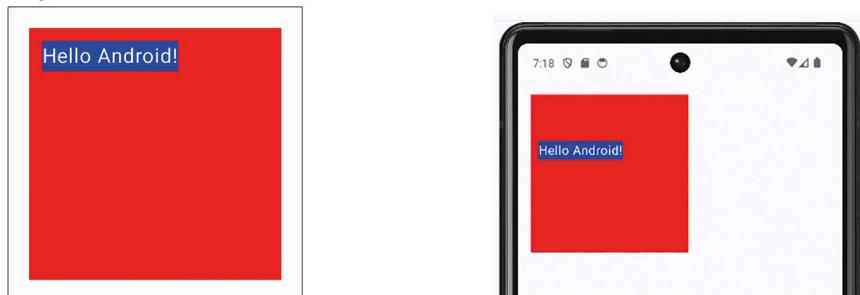


Figura 228

fillMaxSize()

O modificador fillMaxSize() faz com que o composable ocupe todo o espaço disponível em seu elemento-pai.

Vamos substituir o padding do Text por fillMaxSize().

```
Box (
    modifier = modifier
        .padding ( 16 .dp)
        .size ( 200 .dp)
        .background ( Color .Red)
) {
```



```
    Text(
        text = "Hello $name!",
        color = Color.White,
        modifier = modifier
            .fillMaxSize()
            .background ( Color .Blue)
    )
}
```

Figura 229

Unidade II

Depois, vamos substituir o padding do Box:

```
Box ( -  
    modifier = modifier  
    .fillMaxSize()  
    .size ( 200 .dp)  
    .background ( Color .Red)  
) {  
  
    Text(  
        text = "Hello $name!",  
        color = Color.White,  
        modifier = modifier  
        .padding ( 1 .dp)  
        .background ( Color .Blue)  
    )  
}
```

Figura 230

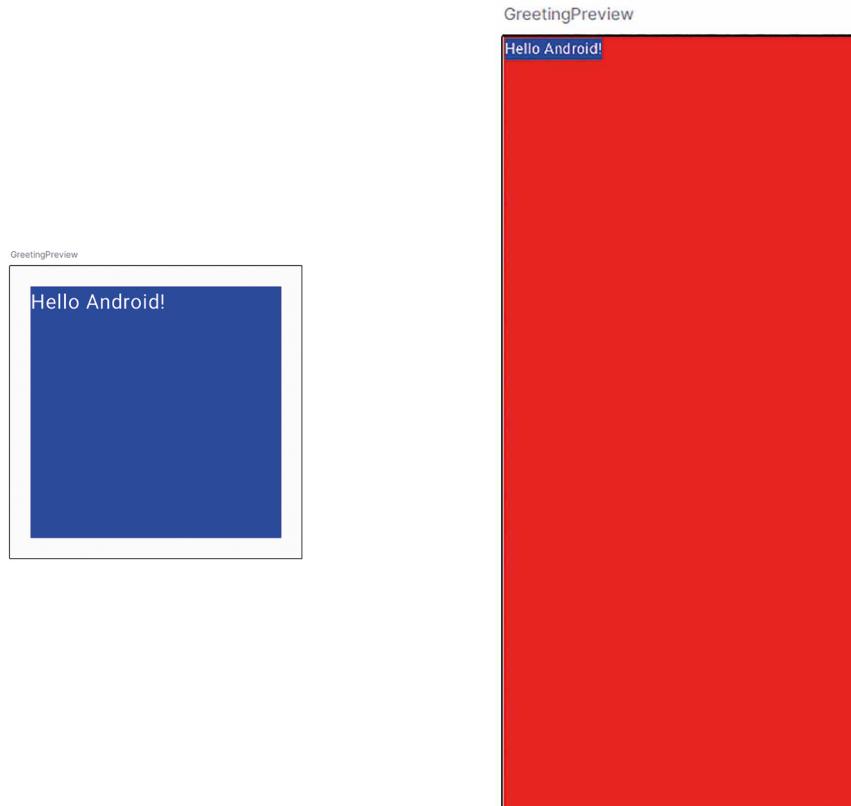


Figura 231

No primeiro caso, o texto ocupou a caixa inteira e, no segundo, a caixa ocupou a tela inteira.

wrapContentSize()

O modificador `wrapContentSize()` ajusta o tamanho do composable com base no seu conteúdo, envolvendo-o em seu torno.

```
Box ( modifier = modifier
    .wrapContentSize(alignment=Alignment.Center)
    .background ( Color .Red)
) {
    Text(
        text = "Hello $name!",
        color = Color.White,
        modifier = modifier
            .padding(1.dp)
            .background(Color.Blue)
    )
}
```

Figura 232

O modificador `wrapContentSize()` ajusta o tamanho do composable para se adaptar ao seu conteúdo, alinhando-o conforme especificado (por exemplo, Center). Este modificador deve ser aplicado antes dos modificadores de tamanho para garantir o comportamento de encapsulamento correto.

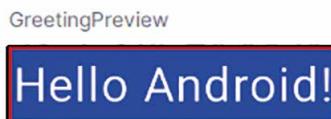


Figura 233

size()

O modificador size() define um tamanho fixo para o composable, substituindo qualquer tamanho padrão ou influenciado pelo elemento-pai.

```
Box ( modifier = modifier
    .size(200.dp)
    .background ( Color .Red)
) {
    Text(
        text = "Hello $name!",
        color = Color.White,
        modifier = modifier
            .padding(1.dp)
            .background(Color.Blue)
    )
}
```

Figura 234

O modificador size(200.dp) define explicitamente as dimensões do composable, tornando-o exatamente 200 dp por 200 dp. Sempre aplique modificadores de layout, como preenchimento ou alinhamento, antes de size() para garantir que eles afetem o composable corretamente.

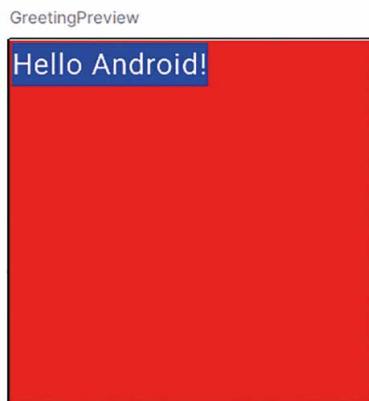


Figura 235

offset()

O modificador offset() desloca a posição do composable por uma quantidade especificada, sem afetar o layout ao redor.

```
Box ( modifier = modifier
    .offset(x = 50. dp, y = 30. dp)
    .size(200.dp)
    .background ( Color .Red)
) {
    Text(
        text = "Hello $name!",
        color = Color.White,
        modifier = modifier
            .padding(1.dp)
            .background(Color.Blue)
    )
}
```

Figura 236

O modificador offset($x = 20.dp, y = 10.dp$) desloca a caixa vermelha interna sem alterar seu tamanho ou afetar o fluxo geral do layout. Para um posicionamento preciso, o offset() deve ser aplicado antes dos modificadores de tamanho e desenho.

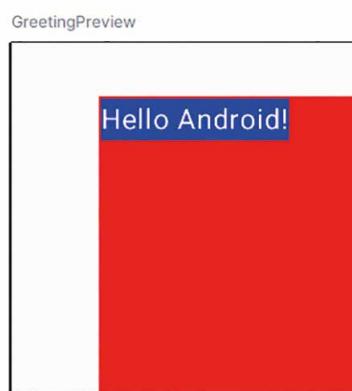


Figura 237

align()

O modificador align() define o alinhamento de um elemento dentro do seu layout-pai.

```
Box (
    modifier = modifier
        .size(200.dp)
        .background ( Color .Red)
) {
    Text(
        text = "Hello $name!",
        color = Color.White,
        modifier = modifier
            .align(Alignment.BottomEnd)
            .padding(1.dp)
            .background(Color.Blue)
    )
}
```

Figura 238

O modificador align(Alignment.BottomEnd) posiciona o elemento composto em um local específico dentro de seu layout-pai, como a extremidade inferior. Para garantir o efeito de layout correto, é crucial aplicar o align() no início da cadeia de modificadores.

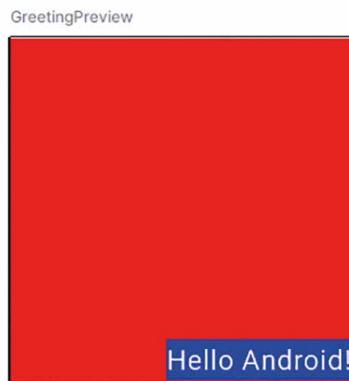


Figura 239

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

weight()

O modificador weight() é utilizado em layouts Row ou Column para distribuir o espaço entre os filhos com base no peso de cada um.

Vamos criar uma linha com três Box.

```
Row(  
    modifier = modifier  
        .fillMaxWidth()  
        .height( 50. dp)  
) {  
    Box(  
        modifier = Modifier  
            .weight( 1 )  
            .fillMaxHeight()  
            .background(Color.Red)  
)  
  
    Box(  
        modifier = Modifier  
            .weight( 1 )  
            .fillMaxHeight()  
            .background(Color.Blue)  
)  
  
    Box(  
        modifier = Modifier  
            .weight( 1 )  
            .fillMaxHeight()  
            .background(Color.Green)  
)  
}
```

Figura 240

O modificador weight permite que os filhos de Row ou Column compartilhem o espaço disponível proporcionalmente com base em seus valores de peso. Para garantir cálculos de layout corretos, é essencial aplicar o weight() antes dos modificadores de desenho, como background().



Figura 241

Modificadores de design

Os modificadores de design determinam como um composable é visualmente renderizado na tela, definindo aspectos como cor de fundo, bordas, sombras e recortes.

background()

O modificador background() adiciona uma cor de fundo ou uma forma a um elemento composable.

```
Text(  
    text = "Hello $name!",  
    color = Color.White,  
    modifier = modifier  
        .align(Alignment.Center)  
        .padding(1.dp)  
        .background(Color.Blue,  
            shape= RoundedCornerShape(16.dp))  
)  
)
```

Figura 242

O modificador background(Color.Blue, shape = RoundedCornerShape(16.dp)) aplica uma cor de fundo azul a um elemento composable e define sua forma com cantos arredondados de 16.dp. Para garantir que o fundo cubra a área pretendida, é importante aplicar modificadores de layout, como padding antes do background().

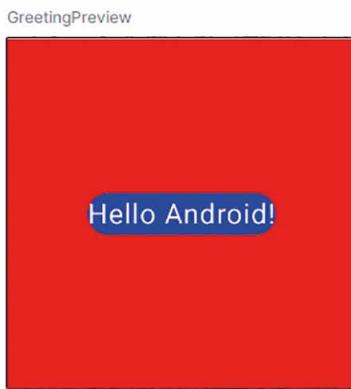


Figura 243

border()

O modificador border() adiciona uma borda ao redor de um elemento composable.

Vamos colocar uma borda preta arredondada em torno do Box:

```
Box ( modifier = modifier .border( 4. dp, Color.Black, RoundedCornerShape( 8. dp )) .size(200.dp) .background ( Color .Red) ) { Text( text = "Hello $name!", color = Color.White, modifier = modifier .align(Alignment.Center) .padding(1.dp) .background(Color.Blue, shape= RoundedCornerShape(16.dp)) ) }
```

Figura 244

O modificador border(4.dp, Color.Black) adiciona uma borda ao redor de um elemento composable com espessura, cor e formato opcionais especificados. Para garantir o espaçamento correto e o posicionamento das bordas, é importante aplicar os modificadores de layout antes do border().



Figura 245

clip()

O modificador clip() recorta um elemento composable em um formato especificado. Vamos recortar a caixa no formato circular:

```
Box {
    modifier = modifier
        .size(200.dp)
        .clip(CircleShape)
        .background ( Color .Red)
    ) {
        Text(
            text = "Hello $name!",
            color = Color.White,
            modifier = modifier
                .align(Alignment.Center)
                .padding(1.dp)
                .background(Color.Blue,
                    shape= RoundedCornerShape(16.dp)
                )
        )
    }
}
```

Figura 246

O modificador clip() confina o composable dentro de uma forma definida, como CircleShape, cortando qualquer conteúdo que ultrapasse os limites dessa forma.

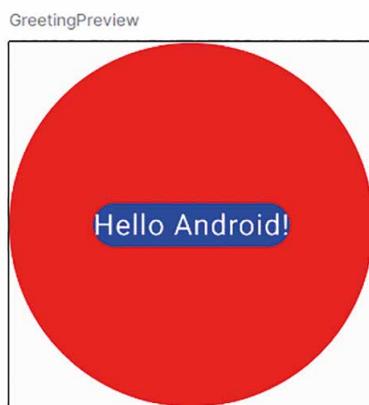


Figura 247

alpha()

O modificador alpha() ajusta a transparência de um elemento composable. Vamos esmaecer um quadrado verde dentro do Box com uma transparência 0.5.

```
Box {
    modifier = modifier
        .size(200.dp)
        .background ( Color .Red)
} {
    Text(
        text = "Hello $name!",
        color = Color.White,
        modifier = modifier
            .align(Alignment.Center)
            .padding(1.dp)
            .background(Color.Blue,
                shape= RoundedCornerShape(16.dp)
            )
    )
    Box(
        modifier=modifier
            .align(Alignment.Center)
            .alpha(0.5f)
            .size(40.dp)
            .background(Color.Green)
    )
}
```

Figura 248

O modificador alpha(0.5f) reduz a opacidade de um elemento composable, permitindo que o fundo ou os elementos sobrepostos fiquem parcialmente visíveis. Para obter o efeito visual correto, ele deve ser aplicado após os ajustes de layout e tamanho.

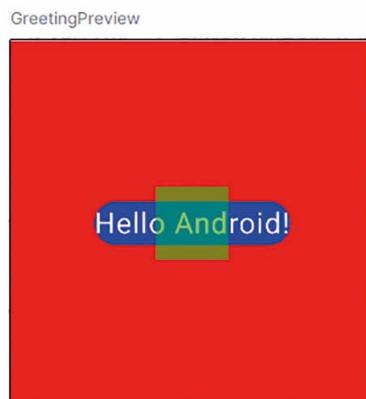


Figura 249

Se alterarmos o alpha para 0.9, temos:

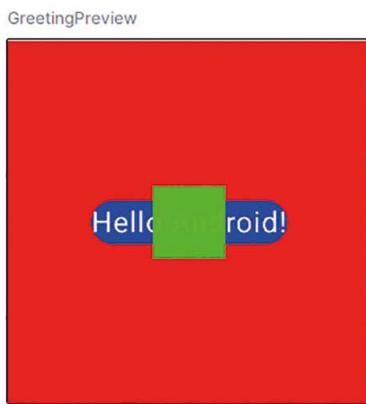


Figura 250

shadow()

O modificador shadow() adiciona um efeito de sombra a um elemento composable.

```
Box (
    modifier = modifier
        .size(200.dp)
        .background ( Color .Red)
) {
    Text(
        text = "Hello $name!",
        color = Color.White,
        modifier = modifier
            .shadow( 18. dp, RoundedCornerShape( 8. dp))
            .align(Alignment.Center)
            .padding(1.dp)
            .background(Color.Blue,
                shape= RoundedCornerShape(16.dp)
            )
    )
}
```

Figura 251

O modificador shadow(18.dp) adiciona profundidade ao criar um efeito de sombra, melhorando a aparência do elemento composable e fazendo-o se destacar. Para refletir o tamanho final da composição, as sombras devem ser aplicadas após as alterações de layout.

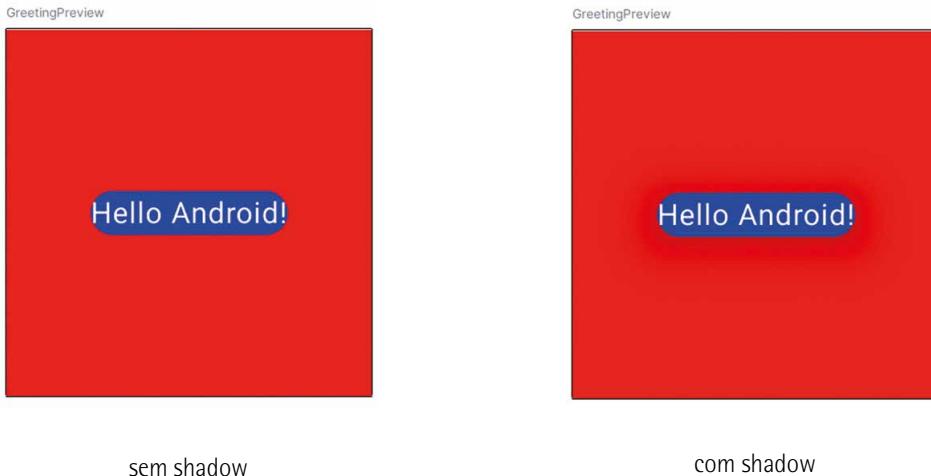


Figura 252

Modificadores de interação

Os modificadores de interação gerenciam entradas do usuário, como cliques, gestos e rolagens, definindo como o composable responde às ações do usuário.

clickable()

O modificador clickable() permite que um elemento composable responda a eventos de clique.

```
Text(
    text = "Hello $name!",
    color = Color.White,
    modifier = modifier
        .clickable { /* Lidar com evento de clique */ }
        .align(Alignment.Center)
        .padding(1.dp)
        .background(Color.Blue,
            shape= RoundedCornerShape(16.dp)
        )
)
```

Figura 253

O modificador clickable() transforma qualquer elemento composable em um componente interativo, permitindo respostas a cliques sem a necessidade de componentes de botão dedicados. Para garantir que a área clicável esteja corretamente definida, é geralmente melhor aplicar os modificadores de layout primeiro.

toggleable()

O modificador `toggleable()` adiciona comportamento de alternância, sendo útil para criar componentes que funcionam como interruptores.

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Box(
        modifier = modifier
            .toggleable(
                value = true,
                onValueChange = { checked ->{
                    /* Lidar com o evento de alteração do estado */
                }
            )
        .size(200.dp)
        .background ( Color .Red)
    ) {
        Text(
            text = "Hello $name!",
            color = Color.White,
            modifier = modifier
                .clickable { /* Lidar com evento de clique */ }
                .align(Alignment.Center)
                .padding(1.dp)
                .background(Color.Blue,
                    shape= RoundedCornerShape(16.dp)
                )
        )
    }
}
```

Figura 254

O modificador `toggleable()` cria um comportamento de alternância, mudando de estado quando interagido, semelhante a uma caixa de seleção ou interruptor. Para garantir que a área interativa esteja corretamente definida, aplique os ajustes de layout antes de adicionar o `toggleable()`.

scrollable()

O modificador scrollable() permite que um elemento composable seja rolável dentro de seu contêiner-pai.

```
Box (
    modifier = modifier
        .size(200.dp)
        .background ( Color .Red)
) {
    Column (
        modifier = modifier.align(Alignment.Center)
            .scrollable(
                state = rememberScrollableState { delta ->
                    delta
                },
                orientation = Orientation.Vertical)
    ) {
        Text(
            text = "Hello $name!",
            color = Color.White,
            modifier = modifier
                .padding(1.dp)
                .background(
                    Color.Blue,
                    shape = RoundedCornerShape(16.dp)
                )
        )
    }
}
```

Figura 255

Este código define uma Column com vários modificadores aplicados. Primeiramente, a Column é centralizada dentro de seu contêiner-pai usando align(Alignment.Center). Em seguida, o modificador scrollable() é aplicado, permitindo que a Column seja rolável verticalmente. O estado de rolagem é gerenciado por rememberScrollableState { delta -> delta }, que registra as mudanças de rolagem (delta) e retorna o valor do delta, possibilitando que o conteúdo dentro da Column seja rolado verticalmente.

Outros modificadores

- **Modificadores de animação:** tornam suas interfaces mais dinâmicas, animando propriedades como tamanho, posição e opacidade:
 - **animateContentSize():** anima a mudança de tamanho de um botão ao ser pressionado.
 - **graphicsLayer():** cria um efeito de rotação em um card ao passar o mouse por cima.

- **Modificadores de gestos:** capturam e respondem a gestos como toque, deslizar e arrastar:
 - `pointerInput()`: detecta um gesto de pinça para ampliar ou reduzir uma imagem.
 - `nestedScroll()`: permite rolar uma lista dentro de um card que também pode ser rolado.
- **Modificadores de foco:** gerenciam o foco do teclado e outras entradas:
 - `focusable()`: torna um campo de texto focável, permitindo que o usuário digite nele usando o teclado.

6.4.8 Gerenciamento de espaço e alinhamento

O Arrangement define como os componentes são distribuídos ao longo do eixo principal de um layout (verticalmente em um Column e horizontalmente em um Row). Além disso, ele controla o alinhamento dos componentes no eixo perpendicular (horizontalmente em um Column e verticalmente em um Row). Os parâmetros `verticalArrangement` e `horizontalArrangement` são usados para definir o espaçamento no eixo principal, enquanto `horizontalAlignment` e `verticalAlignment` controlam o alinhamento no eixo perpendicular.

Aqui estão algumas implementações com as opções mais comuns:

- **Arrangement.Start**: posiciona todos os elementos encostados à margem inicial (esquerda em um Row, topo em um Column).
- **Arrangement.End**: o oposto do Start, alinhando os elementos à margem final.
- **Arrangement.Center**: centraliza todos os elementos no eixo principal.
- **Arrangement.SpaceBetween**: distribui os elementos de forma que o espaço entre eles seja igual, com o primeiro elemento encostado à margem inicial e o último à margem final.
- **Arrangement.SpaceAround**: similar ao SpaceBetween, mas com um espaço extra nas extremidades, fazendo com que os espaços entre os elementos sejam metade do espaço nas extremidades.
- **Arrangement.SpaceEvenly**: distribui os elementos de forma que o espaço entre todos eles seja igual, incluindo o espaço entre o primeiro elemento e a margem inicial e entre o último elemento e a margem final.

Vamos fazer um exemplo com o conceito. Em primeiro lugar, vamos criar um widget com uma linha (Row) ocupando todo o espaço horizontal com três Box dentro dele:

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
@Composable
fun Caixa(modifier: Modifier=Modifier, arranjo: Arrangement.Horizontal=Arrangement.Start){
    Row(
        modifier = modifier
            .border(width = 1.dp, color = Color.Black)
            .fillMaxWidth()
            .height(40.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement = arranjo
    ) {
        Box(modifier = Modifier
            .size(20.dp)
            .background(Color.Red)
        )
        Box(modifier = Modifier
            .size(20.dp)
            .background(Color.Blue)
        )
        Box(modifier = Modifier
            .size(20.dp)
            .background(Color.Green)
        )
    }
}
```

Figura 256

Quando criamos a função, o seu cabeçalho é:

```
fun Caixa(
    modifier: Modifier=Modifier,
    arranjo: Arrangement.Horizontal=Arrangement.Start)
```

A função Caixa define um componente que aceita dois parâmetros: modifier e arranjo. O parâmetro modifier é declarado.

Modifier: Modifier=Modifier,

Isso significa que a função permite que um parâmetro modifier que armazena um objeto do tipo Modifier seja passado. O valor padrão desse parâmetro é um objeto Modifier simples, por isso usamos = Modifier na assinatura do método.

Ter um valor padrão para um parâmetro permite que qualquer pessoa que chame essa função decida se deseja ou não fornecer um valor para ele. Se alguém fornecer seu próprio objeto Modifier, poderá personalizar o comportamento e a aparência da interface do usuário. Caso contrário, será utilizado o valor padrão, que é um objeto Modifier simples. Essa prática pode ser aplicada a qualquer parâmetro.

Para o parâmetro arranjo:

Arranjo: Arrangement.Horizontal=Arrangement.Start)

Armazenará um objeto do tipo Arrangement.Horizontal, que em caso de ser omitido na chamada do widget Caixa terá o valor de Arrangement.Start. Este parâmetro será utilizado no corpo do widget.

No corpo, temos a estrutura:

```
Row() {
    Box()
    Box()
    Box()
}
```

Uma estrutura de linha com três Box enfileirados:

```
Row(
    modifier = modifier
        .border(width = 1.dp, color = Color.Black)
        .fillMaxWidth()
        .height(40.dp),
    verticalAlignment = Alignment.CenterVertically,
    horizontalArrangement = arranjo
)
```

Define um contêiner horizontal chamado Row, que organiza seus componentes-filhos em uma linha. O modifier aplicado ao Row adiciona uma borda preta de 1 dp, faz com que o Row ocupe toda a largura disponível e define sua altura em 40 dp. Os componentes dentro do Row são alinhados verticalmente ao centro, graças ao parâmetro verticalAlignment = Alignment.CenterVertically. A distribuição horizontal dos componentes é controlada pelo parâmetro horizontalArrangement, cujo valor é determinado pela variável arranjo recebido como parâmetro, que pode ser qualquer tipo de Arrangement.Horizontal.

Cada Box é definido como:

```
Box(modifier = Modifier
    .size(20.dp)
    .background(/* uma cor */)
)
```

O Box tem o tamanho de 20 dp e fundo individual (no caso, vermelho, azul e verde).

No widget Greeting, temos um contêiner vertical chamado Column, que ocupa todo o espaço disponível ('fillMaxSize()') e possui um preenchimento interno de 16 dp ('padding(16.dp)'). Dentro dessa Column, há um componente Text que exibe o texto Arrangement.Start e utiliza o modifier passado para ele. Além disso, há uma chamada para a função Caixa(), que adiciona outro componente dentro da Column. Esse arranjo permite organizar os elementos verticalmente, com espaçamento e alinhamento definidos pelo modifier aplicado à Column (figura 257).

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Column {
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp)
    } {
        Text( text = "Arrangement.Start", modifier = modifier)
        Caixa()
    }
}
```

Figura 257

Observe que a chamada ao widget Caixa() não inclui nenhum parâmetro, o que significa que os parâmetros modifier e arranjo assumem seus valores padrão, Modifier e Arrangement.Start, respectivamente. Assim, temos como resultado:

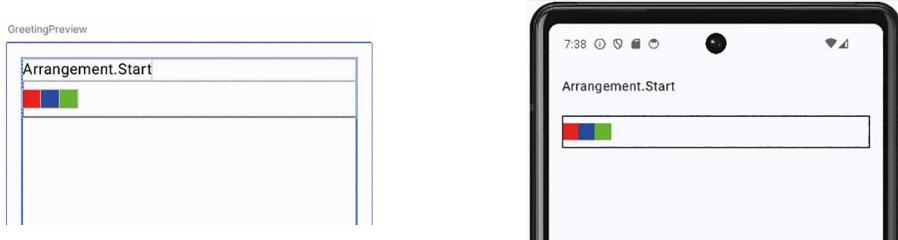


Figura 258

Acrescentaremos então à Column os diversos arranjos e os seus respectivos textos.

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Column {
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp)
    } {
        Text( text = "Arrangement.Start", modifier = modifier)
        Caixa()
        Text( text = "Arrangement.Center", modifier = modifier)
        Caixa(modifier = Modifier, arranjo = Arrangement.Center)
        Text( text = "Arrangement.End", modifier = modifier)
        Caixa(modifier = Modifier, arranjo = Arrangement.End)
        Text( text = "Arrangement.SpaceBetween", modifier = modifier)
        Caixa(modifier = Modifier, arranjo = Arrangement.SpaceBetween)
        Text( text = "Arrangement.SpaceAround", modifier = modifier)
        Caixa(modifier = Modifier, arranjo = Arrangement.SpaceAround)
        Text( text = "Arrangement.SpaceEvenly", modifier = modifier)
        Caixa(modifier = Modifier, arranjo = Arrangement.SpaceEvenly)
    }
}
```

Figura 259

Unidade II

O resultado é uma tela mostrando diversos arranjos horizontais:

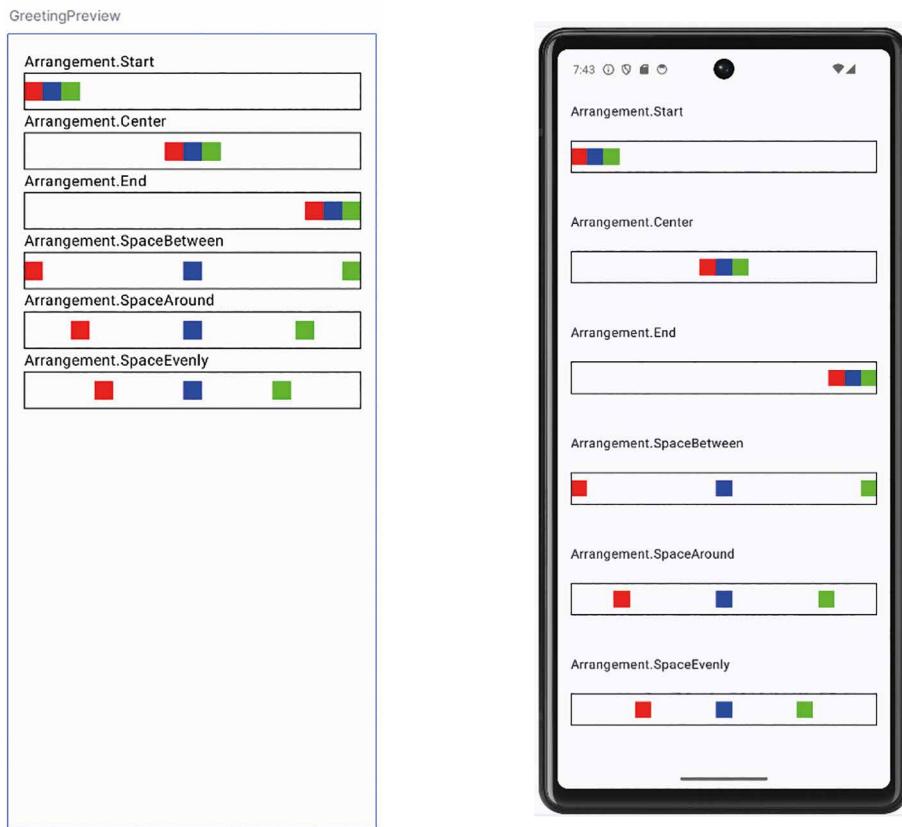


Figura 260

O código final fica:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            MeuExemploAlignmentTheme {
                Scaffold(modifier = Modifier.fillMaxSize()) { innerPadding ->
                    Greeting(
                        name = "Android",
                        modifier = Modifier.padding(innerPadding)
                    )
                }
            }
        }
    }

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

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

```
modifier = Modifier
    .fillMaxSize()
    .padding(16.dp)
) {
    Text( text = "Arrangement.Start", modifier = modifier)
    Caixa()
    Text( text = "Arrangement.Center", modifier = modifier)
    Caixa(modifier = Modifier, arranjo = Arrangement.Center)
    Text( text = "Arrangement.End", modifier = modifier)
    Caixa(modifier = Modifier, arranjo = Arrangement.End)
    Text( text = "Arrangement.SpaceBetween", modifier = modifier)
    Caixa(modifier = Modifier, arranjo = Arrangement.SpaceBetween)
    Text( text = "Arrangement.SpaceAround", modifier = modifier)
    Caixa(modifier = Modifier, arranjo = Arrangement.SpaceAround)
    Text( text = "Arrangement.SpaceEvenly", modifier = modifier)
    Caixa(modifier = Modifier, arranjo = Arrangement.SpaceEvenly)
}
}

@Composable
fun Caixa(modifier: Modifier=Modifier, arranjo: Arrangement.Horizontal=Arrangement.Start) {
    Row(
        modifier = modifier
        .border(width = 1.dp, color = Color.Black)
        .fillMaxWidth()
        .height(40.dp),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement = arranjo
    ) {
        Box(modifier = Modifier
            .size(20.dp)
            .background(Color.Red)
        )
        Box(modifier = Modifier
            .size(20.dp)
            .background(Color.Blue)
        )
        Box(modifier = Modifier
            .size(20.dp)
            .background(Color.Green)
        )
    }
}

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



Resumo

Nesta unidade, introduzimos o Android Studio, incluindo sua instalação, criação de um projeto, navegação pela interface do usuário e uso do emulador para testar aplicativos. Esses tópicos fornecem uma base sólida para desenvolvedores que estão começando a trabalhar com o programa, garantindo que eles possam configurar seu ambiente de desenvolvimento e iniciar seus projetos com confiança.

Em seguida, focamos na Interface Gráfica XML, discutindo os passos iniciais para começar um projeto utilizando Views. Cobrimos a estrutura da interface, utilização de diferentes tipos de Views e como elas podem ser organizadas para criar layouts eficazes e visualmente atraentes. Essas informações são cruciais para entender a construção de interfaces tradicionais em aplicativos Android.

Por fim, exploramos o Jetpack Compose, uma moderna ferramenta de criação de interfaces no Android. Sua introdução foi seguida por tópicos como Material Design, widgets, fundamentos dos layouts, composables, principais layouts em Compose, modificadores de layout e gerenciamento de espaço e alinhamento. Esses assuntos são essenciais para desenvolvedores que desejam aproveitar as vantagens do Jetpack Compose para criar interfaces de usuário mais eficientes e intuitivas.



Exercícios

Questão 1. (IF Sul Rio-Grandense 2021, adaptada) Para iniciar o desenvolvimento de aplicações Android, é necessário fazer a instalação de alguns softwares e realizar algumas configurações.

A respeito do ambiente de desenvolvimento Android, avalie as afirmativas.

I – Android SDK (Software Development Kit) é o software utilizado para desenvolver aplicações no Android, que tem um emulador para simular o dispositivo, ferramentas utilitárias e uma API completa para as linguagens Java e Kotlin, com todas as classes necessárias para desenvolver aplicações.

II – Como existem muitas versões do sistema operacional Android, existe um identificador de cada uma dessas plataformas, que se chama API Level.

III – Gradle é um moderno sistema de gerenciamento de banco de dados para Android.

IV – O Android Studio conta com um utilitário chamado SDK Manager, no qual é possível baixar todas as plataformas do Android e suas documentações, bibliotecas de compatibilidade, bibliotecas do Google Play Services etc.

É correto o que se afirma em:

- A) II, apenas.
- B) I e III, apenas.
- C) I, II e IV, apenas.
- D) II, III e IV, apenas.
- E) I, II, III e IV.

Resposta correta: alternativa C.

Análise das afirmativas

I – Afirmativa correta.

Justificativa: o Android SDK é o conjunto de ferramentas essenciais para desenvolvimento Android, incluindo um emulador, utilitários e APIs completas para Java e Kotlin.

Unidade II

II – Afirmativa correta.

Justificativa: o API Level é um número inteiro que identifica a versão da API do Android. Cada versão do Android tem um API Level único.

III – Afirmativa incorreta.

Justificativa: o Gradle é um sistema de compilação e de empacotamento de projetos (builds), e não um sistema de gerenciamento de banco de dados. Ele é usado para gerenciar dependências, compilar código e criar pacotes de aplicativos (APKs).

IV – Afirmativa correta.

Justificativa: o SDK Manager é uma ferramenta de gerenciamento essencial no Android Studio. Ele permite que os desenvolvedores gerenciem e configurem as versões do Android SDK, bibliotecas, ferramentas e outros componentes necessários para criar, testar e depurar aplicativos Android.

Questão 2. (FGV 2023, adaptada) O analista Marcos está desenvolvendo o aplicativo Android TribunalMovel por meio do Android Studio. Ele criou em TribunalMovel o componente FluxoDois, que exibe uma tela única, com interface gráfica. O FluxoDois constitui, no aplicativo, um novo ponto de entrada para a interação com o usuário. A fim de informar ao Android a presença do novo componente, Marcos precisa declarar o FluxoDois no arquivo de manifesto AndroidManifest.xml.

Para realizar a declaração do FluxoDois, Marcos deve adicionar ao AndroidManifest.xml um novo elemento do tipo:

A) Action.

B) Service.

C) Activity.

D) Receiver.

E) Provider.

Resposta correta: alternativa C.

Análise da questão

Para declarar um novo componente no Android que representa uma tela única com interface gráfica, o analista deve adicionar um novo elemento do tipo Activity no arquivo AndroidManifest.xml.

DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

Activity é o componente do Android que representa uma tela com interface gráfica. Desse modo, cada tela visível ao usuário em um aplicativo Android é uma Activity.

O AndroidManifest.xml é o arquivo no qual todos os componentes do aplicativo (como Activities e Services) são declarados. Ele fornece informações importantes sobre o aplicativo para as ferramentas de build do Android, para o sistema operacional e para a Google Play Store.